Design Of Asynchronous Digital Circuits

by

SUDHIR K. DESAI

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

FEBRUARY, 1997

Design Of Asynchronous Digital Circuits

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Sudhir K. Desai

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

Feb 1997

EE-1997-M-DES-DES

Dedicated

To

My Parents

CERTIFICATE

This is to certify that the work contained in the thesis entitled Design Of Asynchronous Digital Circuits by Sudhir K. Desai has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]
13/02/97

Dr. Subir K. Roy,
Department of Electrical Engineering,
Indian Institute of Technology, Kanpur.

Abstract

The asynchronous approach to realise digital systems has been known for as long as the synchronous approach. However, synchronous approach was the preferred choice because of its simplicity. With advances in VLSI device and fabrication technology resulting in high integration, the synchronous approach for realising digital systems has to deal with the problems of critical path delay, clock skew and increased power dissipation. Besides, technology migration is difficult in the synchronous approach.

These problems can be taken care of by adopting the asynchronous approach. This is the reason why this approach has seen a resurgence specially in the domain of mobile communications and handheld applications.

Asynchronous design approaches are primarily based on the different delay models used. In the present thesis, we develop a new design methodology based on the delay insensitive model for asynchronous circuits which uniformally uses the 2-phase non-return-to-zero transition signalling. We first develop a library of basic modules based on this approach. We also show how designs can be implemented using these elements through illustrative examples. To synthesize designs from their behavioral descriptions in a Hardware Description Language, we need to include additional interconnect elements for point to point and bus interconnection topologies. We study a few of these elements. Our approach is not amenable to synthesis based on the algorithms and approaches employed in the synchronous design paradigm. As such, in the latter part of the thesis, we hand synthesize a few designs from their HDL descriptions to study the various synthesis issues applicable to our approach.

Acknowledgements

I express my deepest gratitude to my guide Dr. S. K. Roy for providing me valuable guidance and encouragement during my stay at I.I.T. Kanpur. The thesis was a major part of the course and doing it under Dr. S. K. Roy was an enriching experience. He was always there to patiently listen to all my problems and offer suggestions. Without his help, this work would never have been completed.

After my graduation, in 1990 , an M.Tech course at IIT Kanpur came like a breath of fresh air. I am thankful to Dr. P. R. K. Rao for giving me this apportunity. The faculty of Electrical Engineering Department was always a source of inspiration. I am also extremely grateful to Vikas Gokhale, Hemant Patil, Jitendra Bhole ,Warsi, Vasu and Bipin for helping me out in all my difficulties.

Amongst the many things which have made my stay memorable are the swimming sessions with Suyog and Milind . It pinches my heart to leave this institution which gave me everything I asked for.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dıgıtal systems can be realısed usıng eıther of the two broad classes of cırcuıts, synchronous and asynchronous Desıgns based on the synchronous paradıgm have been very wıdely used In synchronous systems, we use a clock to coordınate all ıts actıvıtıes The clock ıs dıstrıbuted to varıous parts of the system When the subsystems are wıdely dıstrıbuted ın a chıp, theır synchronızatıon can be very dıffıcult because of the delays assocıated wıth the ınterconnectıon wıres Thıs problem ıs known as the clock skew Furthermore, each ındıvıdual cırcuıt block needs to be desıgned usıng the worst case delays, to ensure that the worst case delay ın a cırcuıt block ıs less than a clock perıod Thıs leads to a very conservatıve desıgn based on the worst case performance As the devıce sızes reduce and theır speed of operatıon ıncreases, delays contrıbuted by the ınterconnectıng wıres becomes domınant Thıs worsens the problem due to clock skews even further

Asynchronous systems, on the other hand, do not use any clock for coordınatıng theır ınternal operatıons Thıs completely elımınates the problem due to clock skews It also results ın desıgns havıng an average case performance ınstead of a worst case performance In asynchronous systems, actıve sıgnals are confıned to the vıcınıty correspondıng to elements ınvolved ın carryıng out a computatıon unlıke ın synchronous systems where transıtıons arısıng out of the clock sıgnal takes place ın even ıdle elements Thıs results ın a lower power consumptıon for asynchronous systems Also, individual circuit blocks can be designed separately without having to satisfy any global timing constraint This eases the timing issues and rarely used portions of the circuit can be left unoptimised without hampering the overall performance to a great extent

However, asynchronous circuits are more difficult to design as compared to their synchronous counterparts Moreover, the circuit implementations can be quite complex and are generally cumbersome Therefore, there exists a need for automating the design of digital systems based on the asynchronous approach Many methodologies exist for the design of asynchronous systems [1] All of them use an underlying delay model and can be classified on the basis of the delay model they use The major delay models are as follows

- Bounded delay model
- Delay Insensitive model
- Quasi-delay-insensitive model
- Speed independent model

The bounded delay model, also called the Huffman's model assumes that delay in all the circuit elements and wires are known or at least bounded While designing circuits using this model, extreme care has to be exercised to avoid hazards [2, 3] One of the ways to avoid hazards is not to allow multiple input changes Hazards, due to single input changes are eliminated by adding redundant circuit blocks [3] However, this degrades the final performance of an implementation based on this model None of the methodologies available for this delay model addresses the system design issues Hence, synthesis methods based on this model have not evolved

The delay insensitive model assumes that the delays in both the circuit elements and wires are finite but unbounded Because this delay model offers the least restrictions on delay assumptions, the circuit design based on this model is very attractive But it has been found that the class of purely delay insensitive circuits realised using the classical circuit elements like AND, OR, etc is extremely limited [4]

In speed independent model, it is assumed that while the gate delays are unbounded, the wire delays are negligible The quasi-delay-insensitive model is based on the delay insensitive model where both the gate and wire delays are unbounded However, it restricts the wires connecting fanout elements to have the same delay This is known as the isochronic fork Thus isochronic forks are the forking wires where the delays in all the forking wires is nearly identical The quasi-delay-insensitive model can be seen to satisfy the assumptions made in the speed independent model

As compared to the delay insensitive model, the speed independent and the quasi-delay-insensitive model offer more implementation alternatives But the delay assumptions are difficult to realise The delay assumption in speed independent circuits is no longer valid for all the technologies, e g , FPGA's, where wire delays often dominate It is also not valid for large systems Also the implementation of isochronic forks in the quasi-delay-insensitive circuits can be difficult to realise, especially when the forking ends are on different chips Considering this, the circuit design under delay insensitive model is the most attractive option

Ebergen proposed a synthesis method for designing delay insensitive circuits which is based on the trace theory [5] It uses circuit elements like C element, toggle, merge, wire, etc Brzozowski and Ebergen [6] proved the C element and toggle blocks can not have a DI implementation using conventional level sensitive gates such as AND, OR, NOT, etc Leung and Li [7] have recently proposed a set of properties to characterize the DI behaviors of any circuit elements It has also been conjectured that, any basic circuit element used for realising the DI behaviors, can not be delay insensitive internally So, design of delay insensitive systems is possible using modules which are not internally delay insensitive One of the examples is the Q-modules designed by Rosenberger et.el [8]

Though, Ebergen's synthesis methodology provides a good theoretical basis for the design of delay insensitive circuits, there exists problems in its wide applicability The trace theory is very difficult for humans to understand as it is nonintuitive Designing circuits using this methodology forces a designer to think at individual transition levels for each new circuit to be designed This renders describing behaviors of large systems such as complex microprocessor very difficult The motivation for this work arises from the drawback observed above, and is to synthesize delay insensitive asynchronous digital systems from their behavioral description in a Hardware Description Language (HDL) This approach will provide a much simpler and easier route to employ than the one based on the trace theory Further, we assume a 2-phase, non-return-to-zero (NRZ) event driven scheme adopted in [9, 10] In this scheme, data signals are 2-rail A data value of '1' is indicated by a transition on one of its two rails (r_1) and a '0' is indicated by a transition on the other rail (r_0) Transitions, also called events, occuring on both the rails simultaneously imply an invalid data and are not allowed Control signals, on the other hand, are single rail and a transition on a control wire initiates the control operation associated with it

The NRZ transition signalling is used because, it gives better performance than the RZ transition signalling The RZ signalling is used in TITAC work [11], which is a quasi-delay-insensitive microprocessor Consider a register to register data transfer in this microprocessor, as shown in Figure 1 1 Let the data stored in Reg A and Reg B is to be transferred to a functional unit FU The output of FU is then to be transferred to Reg C To carry out this task, controller makes its Request signal logic high This transfers the data in Reg A and Reg B to FU and its output is transferred to Reg C After Reg C receives data, Ack wire is made high to signify the end of operation This constitutes the working phase of the operation This is further subdivided into the working transient (WT) and the working stable (WS) subphases as shown in the figure After this the controller pulls its Request signal low This, in turn, pulls all the signals in the circuit low including Ack This constitutes the idle phase which is again subdivided into the idle transient (IT) and the idle stable (IS) subphases The presence of an idle phase degrades the performance

The abovementioned task of designing and synthesizing asynchronous digital systems necessitates the existence of a library of basic modules designed using the DI model assumption Thus we need modules which can perform basic logic functionalities such as AND, OR, XOR, etc in the adopted signalling protocol Nanda et al [9, 10] use a U-gate to realise all the basic Boolean expressions In the same

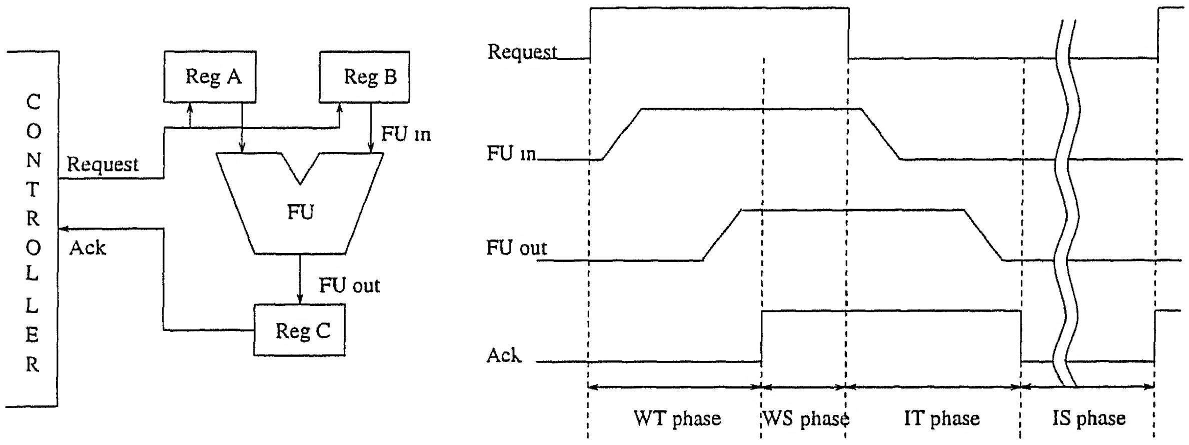


Figure 1 1 a) Register to register event driven data transfer b) Two phase operation

work, a Shift Multiplier is designed using U-gates and other circuit blocks Conversion between logic levels and transition signals and vice-versa are done with 1to2 Converters and 2to1 Converters respectively However, there is no mention of a consistent way to design and synthesize asynchronous systems Instead, an approach based on the equivalent synthesized synchronous design has been given This can lead to non-optimal implementations as will be shown later

Chapter 2 describes the design of basic blocks which are intended to be present in the library to be used for synthesis In chapter 3, few design examples are illustrated which use the basic blocks described in chapter 2 Interconnection between the various resources is described in chapter 4 Two methods of interconnection are considered, point to point interconnections and the bus structure Three approaches to implement the bus structure are described In chapter 5, various synthesis issues are presented, where we see that the synthesis issues are dominated by the interconnection topology used Three interconnection elements for the point to point topology are proposed, which results in a better implementation of the interconnection network Two design examples, Shift Multiplier and differential equation integrator are presented to justify the synthesis process outlined earlier Chapter 6 concludes the thesis and provides pointers to some issues which could not be addressed in the present thesis

Chapter 2

Basic Modules

Any implementation of digital systems requires a library of basic circuit elements to realise complex Boolean functionalities Resources to store data and to carry out the data transfers between them are needed very often The implementation of the controller, which coordinates the various activities in the system is also realised using the same library of circuit elements We therefore need to create this set of basic circuit elements to define the above library In the design methodology adopted, logic functionalities are realised using U gates [9, 10] The NRZ asynchronous data can be stored in either the asynchronous registers called UReg or its variation, REG1 However, UReg and REG1 are not the only circuit elements In fact, several other storage elements such as A_Demux_Store, Const_Demux_Store and Iter_Var_Store which are more suitable for a particular data type, have been used The controllers have been implemented primarily using C elements [12], Select blocks [9, 10], XOR gates and Event Counters

This chapter starts with the description of UReg and REG1 It is followed by the Event Counters Asynchronous toggle flipflops are described next Finally, a way to ensure a known initial condition in the circuit elements is illustrated

2.1 Asynchronous Register (UReg)

The objective of designing the UReg is to store and transfer out the asynchronous data for the NRZ transition signalling format

Figure 2 1 shows the functional representation and circuit diagram of the UReg It has a 2-rail input *IN*, control signals *Data_store_c* and *Data_out_c* and 2-rail outputs *Data_store* and *Data_out* indicated by $\{DS_1, DS_0\}$ and $\{D_1, D_0\}$ respectively

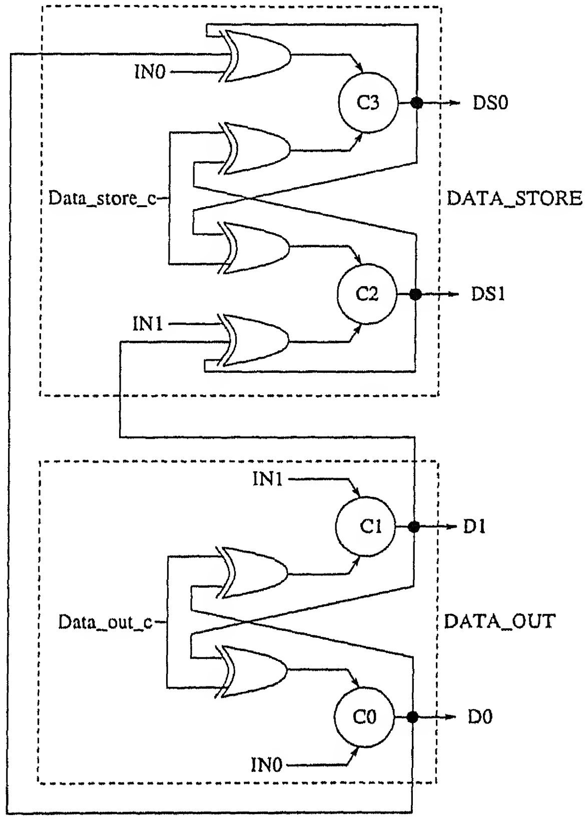


Figure 2 1 Asynchronous Register UReg

Input data is applied at *IN* *Data_store_c* controls the availability of the input to the output terminals *Data_store* It also retains the present input in the register so that the next application of *Data_store_c* will generate the same output on the *Data_store* terminals *Data_out_c*, on the other hand, controls the availability of the input to the *Data_out* terminals, after which, a new input can be applied to the UReg

The circuit can be divided into two blocks, namely, data store block and data shift block, both of which receive the input The data store block consists of Muller C elements C_2 and C_3 with XOR gates connected to them Feedback from the data shift block is also connected to the XOR gate inputs in this block The other signals in this block are the control signal *Data_store_c* and the corresponding 2-rail output signal *Data_store* Muller C Elements C_0, C_1 and XOR gates connected to it along with *Data_out_c* and *Data_out* terminals constitute the data shift block

Circuit Operation

Suppose bit 1 is applied at the input, i e , a transition occurs on *IN1* Therefore, one input of C_1 and C_2 see a transition Now the application of *Data_store_c* will place transitions on the inputs of C_2 and C_3 As both the inputs of C_2 have received a transition, C_2 will fire to produce an output transition at DS_1 Thus, the application of *Data_store_c* again reproduces the input on *Data_store* terminals The transition on D_1 is fed back to C_2, in order to be able to store the data The same transition is used to cancel the predeposited transition on the input of C_3 Thus UReg is taken to the same condition that was present, before the application of *Data_store_c* signal with respect to the given input New *Data_store_c* can be applied to induce an identical set of events, producing a new transition on DS_1

To transfer the stored data corresponding to the above input, *Data_out_c* is applied The transition on *Data_out_c* is deposited on one of the inputs of C_0 and C_1 C_1 will produce a transition on D_1, indicating that the data has been shifted out Also, the same transition cancels the predeposited *Data_out_c* transition on C_0 and the transition corresponding to the input data on the input of C_2 in the data store block With this, there will exist no transition in the UReg, which then will be ready to receive the next data bit

Simultaneous changes in the input *IN* and one of the control signals is valid only when no transition exists in the UReg, i e while using the UReg for the first time and after each application of *Data_out_c* Of course, only one input rail can have a transition at a time and simultaneous occurences of transitions on both the control signals are prohibited No new data can be applied before the previous data has been shifted out of the UReg Also, successive application of *Data_store_c* should be separated by subsequent occurences of *Data_store*, to fulfill the "cause-reaction" relationship, which characterizes the delay insensitive way of functioning

2.1.1 REG1

A register REG1 has a 2-rail input *Inp*, two control signals *Dt_out_c* and *Clr* and a 2-rail output *R_out* It also has a *ClrAck* output A transition on *Clr* input terminal removes the old data out The completion of this is indicated by a transition on *ClrAck* output A transition on the *Dt_out_c* control input on the other hand, produces the data in the register on the *R_out* terminals The data is retained by the register in this process The register is designed such that on global reset, it will be loaded with a value of 0

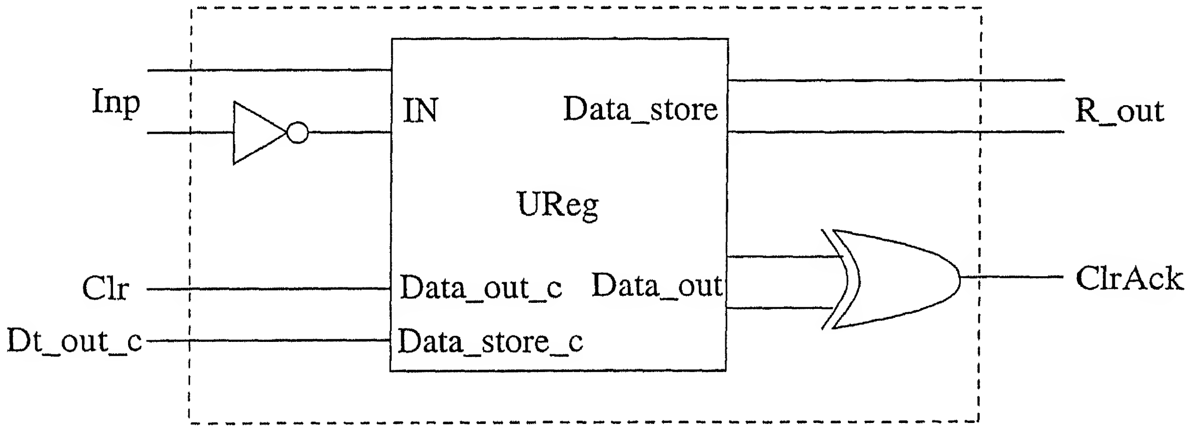


Figure 2 2 Asynchronour register REG1

A REG1 is realised using a UReg as shown in Figure 2 2 The inverter at its input ensures a presence of a data bit 0 on the application of a global reset A transition on *Clr* input terminal is applied to the *Data_out_c* input terminal of UReg and this shifts the contents of UReg on its *Data_out* terminals The XOR gate converts this data to an acknowledge signal *ClrAck* signifying the completion of operation It can be seen that, a transition on *Dt_out_c* input terminal of the REG1 produces the stored data on *R_out* terminals and is also retained

2.2 Event Counters

A modulo n Event Counter has one single rail input *EventIn* and n single rail outputs O_1 through O_n In this, the ith input transition, ($i < n$), creates a transition on output O_i After the nth input transition, the cycle repeats

One of the possible implementation is as shown in the Figure 2 3 The inverters cause an initial transition to be deposited on the those inputs of the C elements, to which they are connected after a global reset

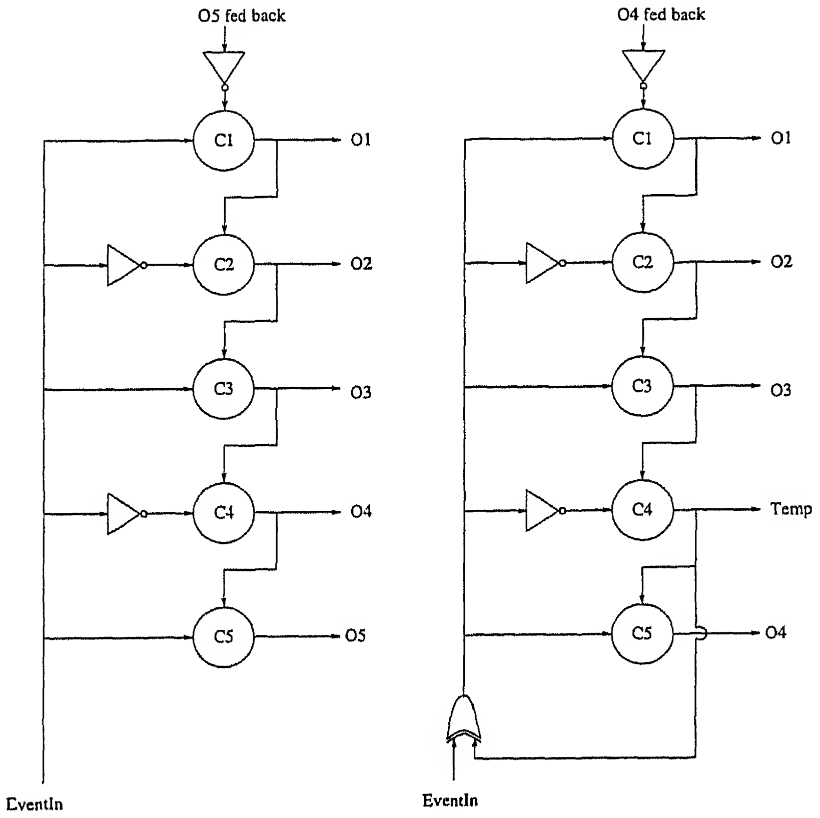
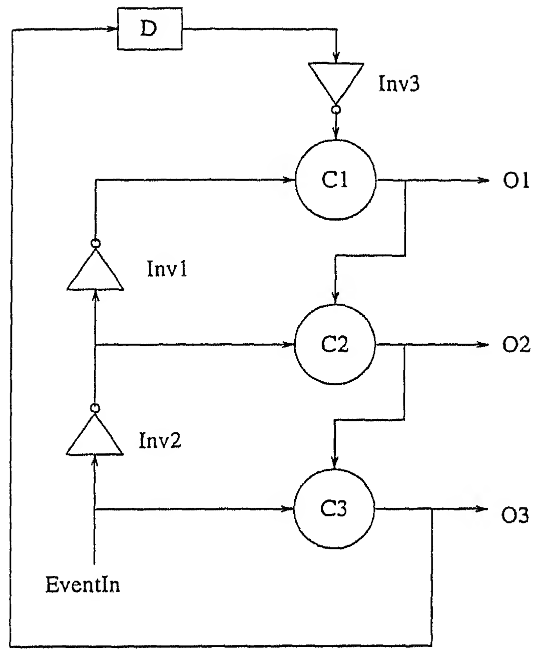
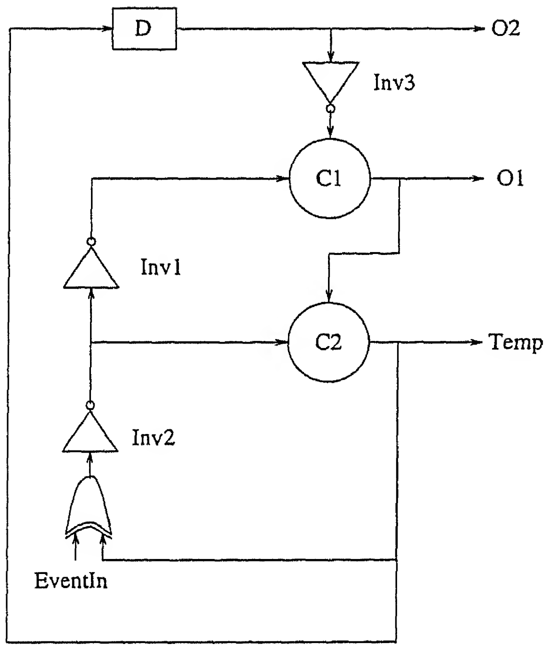


Figure 2 3 Event Counter Implementation 1

Each odd numbered input transition on *EventIn* will place a transition on one input of an odd numbered C element, viz , C_1, C_3, C_5 It will also cancel the input transition on even numbered C elements, viz , C_2 and C_4 connected to *EventIn* through the inverters Similarly, each even numbered transition on *EventIn* will place a transition on an input of even numbered C elements and will cancel the



Delay from EventIn to input of C1 through O3
>
Delay from EventIn to input of C1 through Inv1 and Inv 2



Delay from Temp to input of C1 through D
>
Delay from Temp to input of C1
through XOR gate, Inv1 and Inv2

Figure 2 4 Event Counter Implementation 2

predeposited transition on corresponding inputs of odd numbered C elements Thus, at any given instant, only a single C element has a transition on that input which is not connected to *EventIn*

Initially, C_1 has a transition on one of its input The first transition on *EventIn* creates a transition on output O_1 This transition is issued to the input of C_2 As the transition on the other input of C_2 has been canceled by the previous transition on *EventIn*, possible hazardous event on O_2 is avoided The next transition on *EventIn* will enable C_2 to create an event on O_2 The transition on the nth output is fed back to the input of C1 so that a subsequent transition on *EventIn* will create a transition on O_1

Between any two successive transitions at the output of any C element, it receives n transitions on its input connected to *EventIn* Out of these n transitions, the first transition is utilized to create an event on the output of the corresponding C element and is thus consumed The remaining $n-1$ transitions should not result in any transition on this input of the C element Thus n is forced to be an odd number

Hence, if the number of outputs is even, the circuit needs to be modified so that the above condition is satisfied Such a modified circuit for four outputs is shown in Figure 2 3 An intermediate output indicated by *Temp* is used to create an extra transition at the input through an XOR gate The nth (even) transition creates a transition on *Temp*, which is fed back to create an additional event, fulfilling the condition that the input should receive odd number of transitions This in turn produces the final output transition

Alternatively, *Temp* can be used as O_4, eliminating the use of C_4 However, this imposes the following delay constraint Delay from *Temp* to input of C_1 connected to it through an inverter should be greater than that from *Temp* to the other input of C_1 through an XOR gate

Let a transition on *EventIn* create a transition on output of any C element C_i This transition on *EventIn* must cancel the transition on the input of C_{i+1} connected to *EventIn*, before C_{i+1} receives a transition on its second input generated at the output of C_i Thus the delay in the path from *EventIn* to input of C_{i+1} through output O_i should be greater than that in the path from *EventIn* to the other input of C_{i+1}

All these local delay constraints can be avoided by modifying the circuit such that only one delay constraint needs to be satisfied We ensure this as follows A transition to the input of C element Ci connected to *EventIn* is issued only after a transition on input of C_{i+1} connected to *EventIn* is canceled The modified circuit is shown in Figure 2 4

The abovementioned condition is violated for Cn, as a transition to its input connected to *EventIn* can be issued before the respective transition on $C1$ is canceled This is corrected by the delay element D, satisfying the delay constraint indicated in the figure

Delay constraints for realisation of Event Counter of Figure 2 3 and Figure 2 4 can be eliminated by using a variation of Muller C element This element can be realised as shown in Figure 2 5 Assuming a delay constraint such that the delay in the lower input path is less than that in the upper path, this block can be rendered into an Delay Insensitive (DI) block Thus, event counter can have a quasi-delay insensitive realisation This is shown in the Figure 2 5

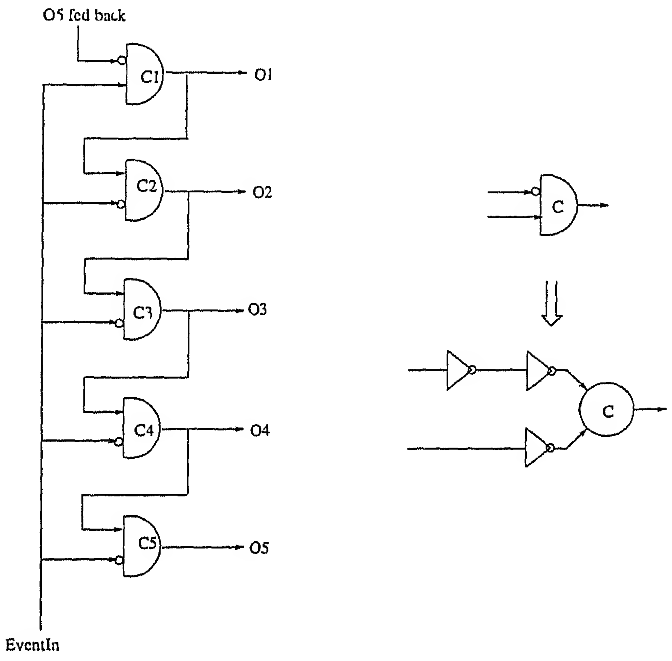


Figure 2 5 Event Counter Quasi-delay-insensitive realisation

The complexity of each of the structures discussed above is linear with the number of outputs required in a mod n counter For every increase in n by 1, at most one C element and an inverter is needed

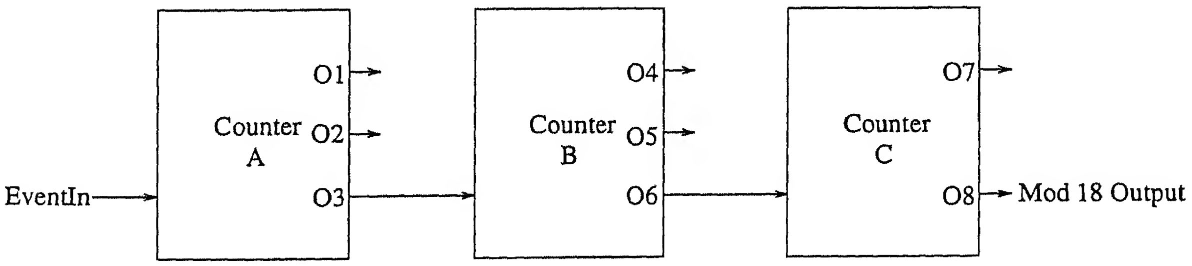


Figure 2 6 Mod-18 Event Counter

If all n outputs of a modulo n Event Counter are not needed, the circuit can be made compact by realising it using Event Counters with smaller number of outputs

For example, a modulo 18 counter is realised using a single modulo 2 and two modulo 3 counters as shown in Figure 2 6 Counter A, which is a modulo 3 counter, responds to every event on its input, *EventIn* It recycles after three consecutive events on *EventIn* Thus, output O_3 of Counter A shows a transition on the third, sixth, nineth, event on *EventIn* and so on This output is connected as an input to Counter B, which is a modulo 3 counter Therefore, output O_6 of this counter, will respond to every ninth transition on *EventIn* It can be seen that, O_8 responds to every eighteenth transition on *EventIn* The response of other outputs can be easily verified

2.3 Asynchronous Conditional Toggle FlipFlop (A_CTff)

The objective of an A_CTff is to have an asynchronous counterpart of synchronous toggle flipflop The functional representation and circuit realisation is as shown in Figure 2 7 It has two double rail inputs *T* and *Init*, two single rail control signals, *Next_state* and *Clear* It also has two double rail outputs, *Q0* and *Cascade*

Clear is used to remove a data bit residing in the flipflop in the form of a transition The completion of a clear operation is indicated by a transition on *ClrAck* Depending on binary value present in the *T* input, a transition on *Next_state* toggles the current contents of the flipflop or retains it After the application of a global reset or a *Clear*, no transition exists in the flipflop Initialization inputs can be applied to the *Init* terminals only after the flipflop has attained the above state Data in the Init input is immediately made available on Q_0 output

Now for every transition on *Next_state* input, if the input *T* is 1, the existing bit in the flipflop is toggled and relevant transition is produced on the Q_0 terminals On the other hand, if *T* is 0, the same data bit is produced on Q_0 terminals Two rail signal *Cascade* is used to cascade the flipflops in applications such as the two rail counter described in the following chapter

The A_CTff shown in Figure 2 7 is realised using two U gates and 5 XOR gates A data bit on *Init* input is stored in U2 Application of a transition on *Clear* terminals

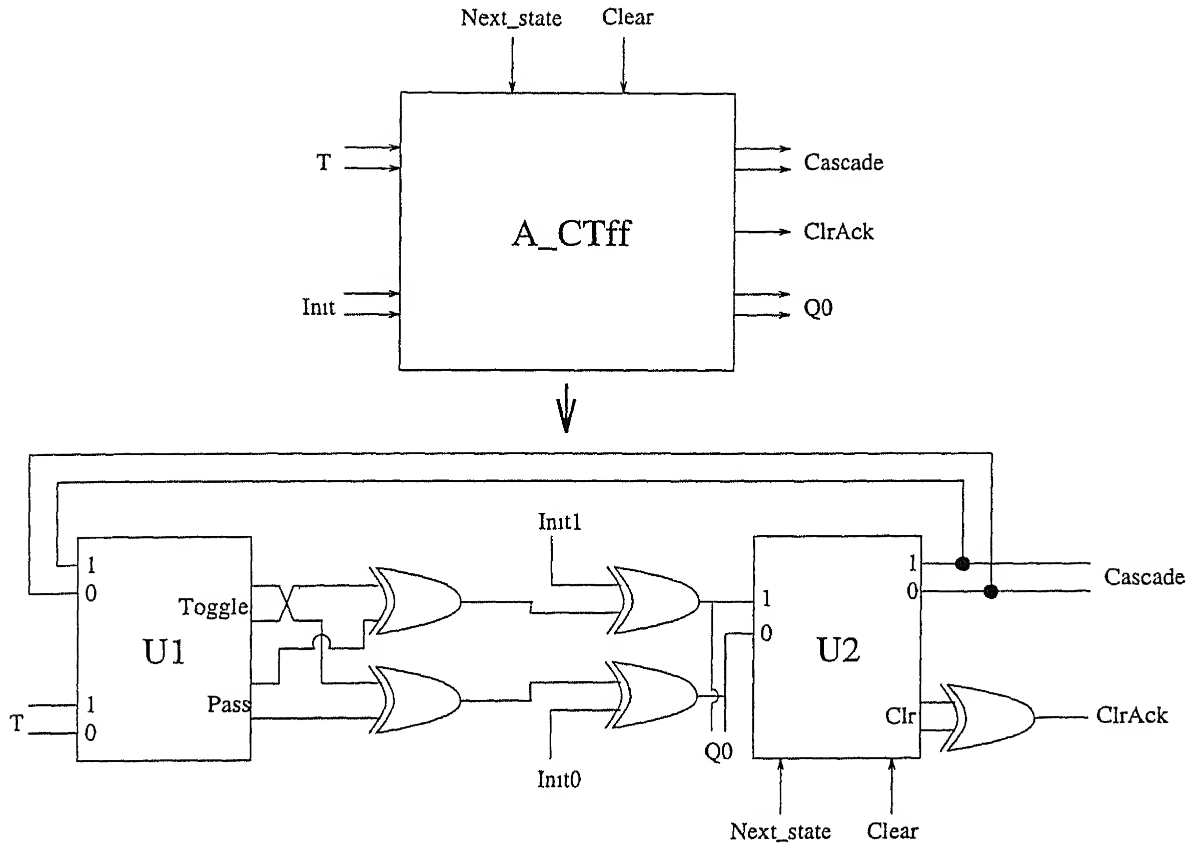


Figure 2 7 Asynchronous Conditional Toggle FlipFlop

transfers the stored data to *Clr* output of U2 This is the process by which any stored data is removed from the flipflop To signify the completion of the data removed from the flipflop, we generate a *ClrAck* using an XOR gate

Application of a transition on *Next_state*, on the other hand, moves the data bit to *Cascade* terminals These terminals are fed back as an input to U1 Assume T is 1 Then the initial data bit stored in U2 and also present on *Cascade* terminals through an event on the *Next_state* input terminal, is shifted to the *Toggle* terminals of U1 While if T is 0, it is shifted to the *Pass* terminals of U1 The *Toggle* and *Pass* terminals are connected along with *Init* inputs such that a data bit produced on the *Toggle* terminals is inverted at Q_0, while that produced on the *Pass* terminals goes through unchanged

The processed data with respect to T is available at Q_0 It is also stored in U2

Thus we see that, in response to an event on *Next_state* input, a stored data is moved out of U2 through *Cascade* terminals Depending on the value of T, it is processed by U1 and the processed data bit is fed back to U2 This constitutes a loop of events for which there exists a possibility of hazards In fact, we can see that, in the above implementation there exist no hazards because the processed bit received by U2 is generated using the one which was stored earlier Thereby, a temporal dependence of the new event on an earlier event precludes generation of any hazard After this the flipflop is ready to receive next control signal The necessary acknowledge to the controller can be generated using Q_0 terminals

For the proper functioning of A_CTff, the following is implicit

1 The control signals *Next_state* and *Clear* cannot be applied simultaneously

2 *Next_state* or *Clear* can be applied only if data is residing in the flipflop

3 Each occurence of *Next_state* should be supported by an application of T in order to complete the desired operation

4 Reset or *Clear* should be followed by a new data on *Init*

2.4 Asynchronous Toggle FlipFlop (A_Tff)

Compared to the A_CTff, A_Tff has the same set of inputs and outputs except for the absence of input T The only functional difference lies in the fact that, on every transition on the *Next_state* input, the value of the initial data stored in U is toggled by inverting the same, and is made available at the Cascade terminals as shown in Figure 2 8 This implementation uses only a single U gate which is functionally equivalent to U2 of A_CTff

2.5 Modified 1To2 Converter

The 1to2 converter has a logic level input C It receives a 1-rail control signal *Read* which is internally referred to as X It has a 2-rail output denoted by $\{Z_1, Z_0\}$ A

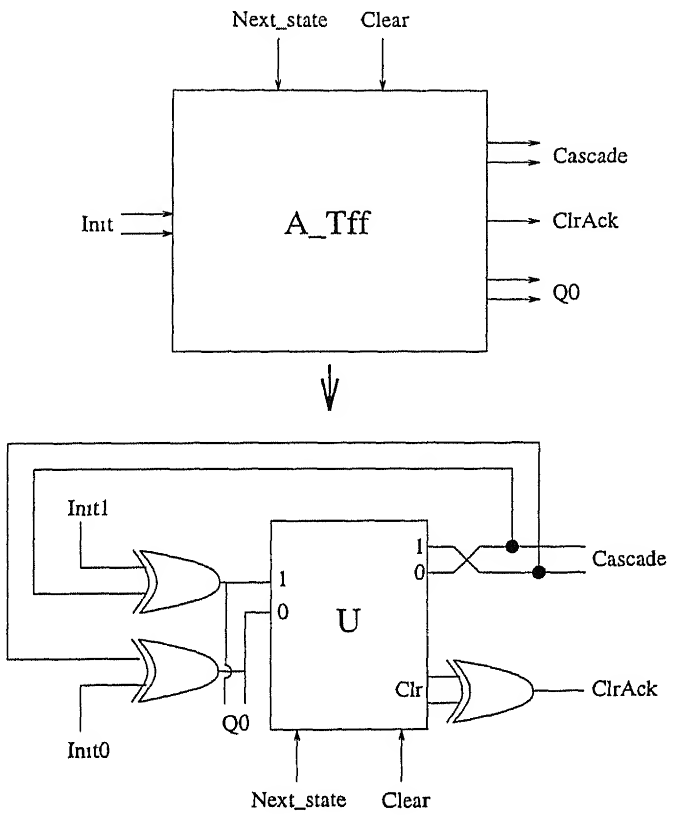


Figure 2 8 Asynchronous Toggle FlipFlop

transition on the *Read* terminal creates a transition on Z_1 if C is 1, else, a transition is created on Z_0 The circuit is realised using two subcircuit blocks BR and BF shown in Figure 2 9a

The block BR creates a transition on its output Z, if the input C is high (level sensitive) and a rising transition is applied to the X terminal This can be understood as follows Let C be high and X be low The complement of Z is stored by C_1, while C_2 retains its previous value When X goes high, the value of C_2 becomes the complement of C_1 which is Z Also C_1 gets isolated from Z since X is high This in turn, creates a transition on Z In the block BF, a transition on Z is created for a falling transition on X, provided C has a logic level 1

The modified 1to2 Converter is realised using these blocks as shown in Figure 2 9b

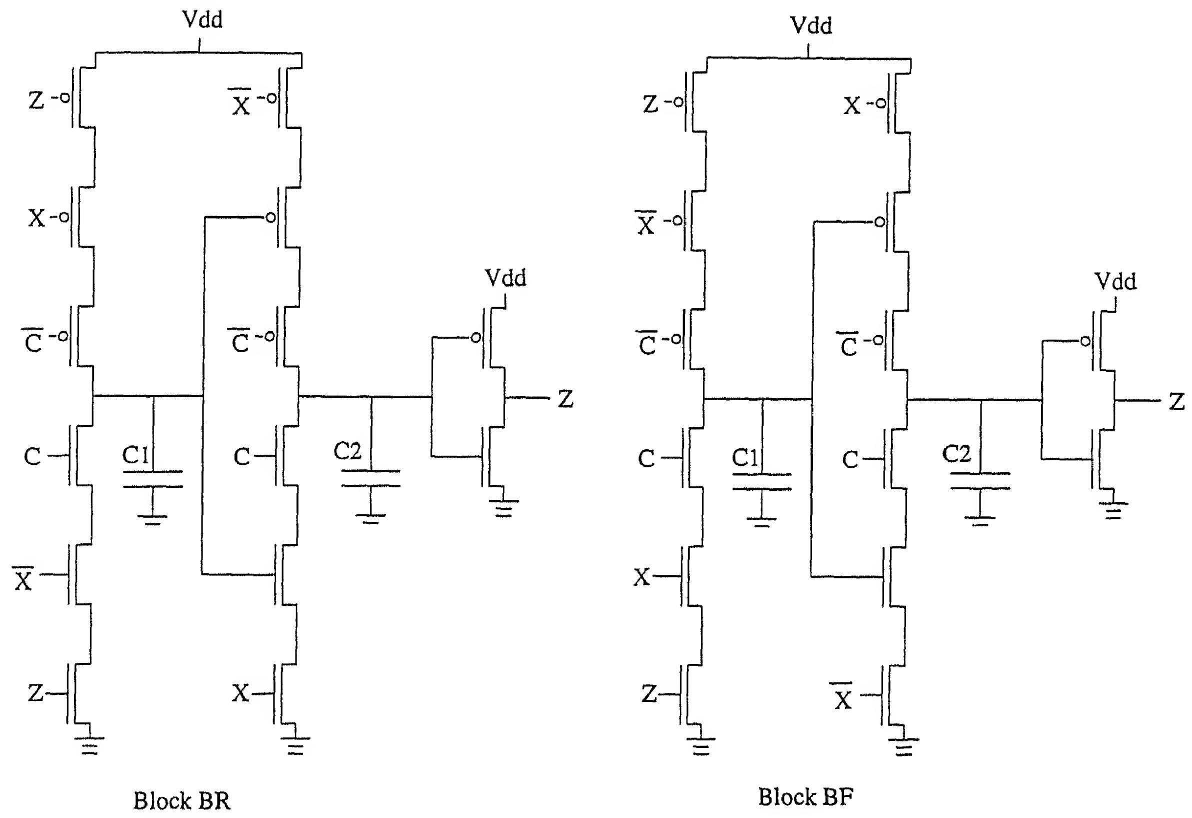


Figure 2 9a Modified 1to2 Converter blocks BR and BF

2.6 Initialization on Reset

All the circuit elements need to be taken to a known initial state on power on and global resets It is assumed that, on the application of global reset, the input and the output terminals of all the C elements are set to a logic value of 0 In order to achieve this, a resettable C element is used Besides the input and output terminals, it also has a *Reset* terminal An application of a logic 1 to the *Reset* terminal forces the output of C element to take a logic value of 0 Two possible implementations are shown in Figure 2 10

As all the basic circuit elements are realised using C elements and XOR gates, an application of logic zero on the *Reset* terminal along with the application of logic zero on all the primary inputs force the circuit elements to the desired initial condition

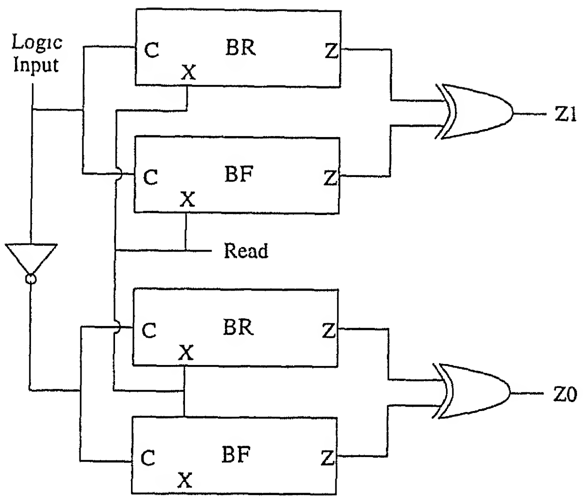


Figure 2 9b Modified 1to2 Converter   Realization using BR and BF

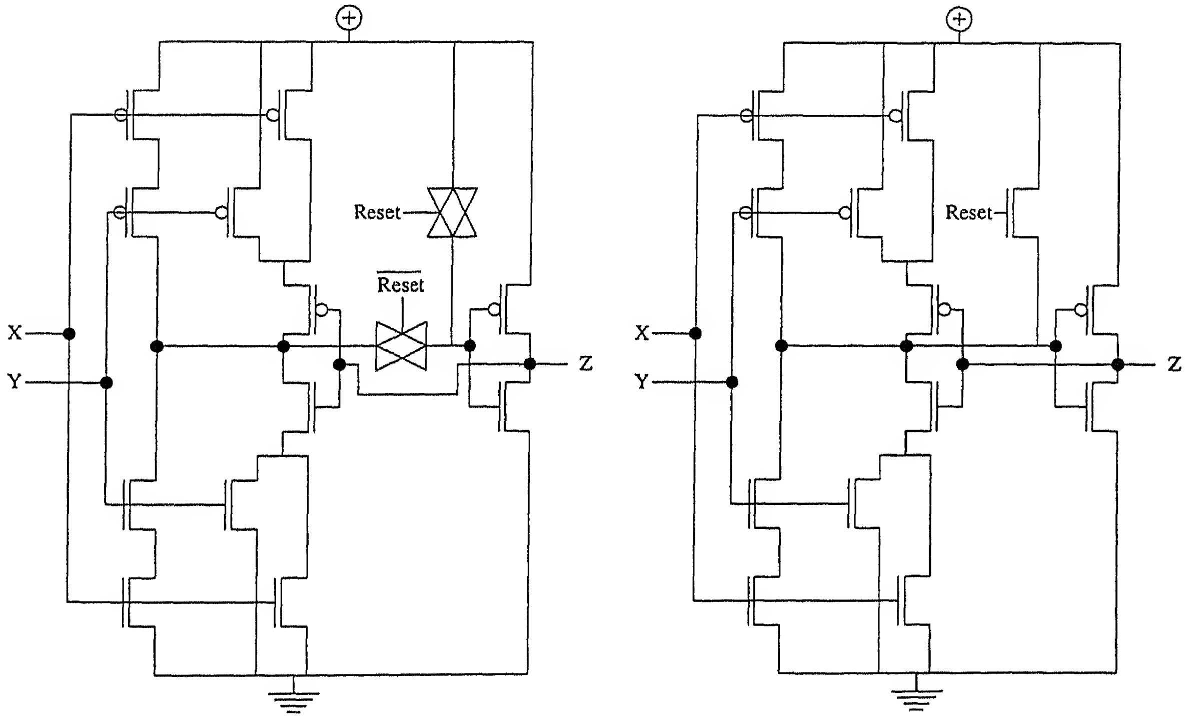


Figure 2 10 Two implementations of the Resettable C Element

Chapter 3

Design Examples

The use of the basic modules described in the last chapter in implementing asynchronous designs is illustrated through three examples The first example is that of a Serial In Parallel Out (SIPO) shift register This is followed by the design of a Polynomial Serial Parallel Multiplier (PSPM) Finally, the design of a 2-rail mod-8 up-down counter based on the A_Tff and A_CTff basic modules is given

3.1 Serial In Parallel Out Shift Register

In many DSP applications, data is applied serially and is processed as it travels on its way to the output Designs based on such a paradigm is illustrated in the next section with the example of a Polynomial Serial Parallel Multiplier (PSPM) This multiplier employs a Serial In Parallel Out (SIPO) shift register described below

The design of PSPM follows the Micropipeline paradigm introduced by Sutherland [12] The only difference in our case being that the data is 2-rail NRZ event driven, as we assume the delay insensitive model instead of the bundled data constraint

The module representation of a 4 bit SIPO asynchronous shift register is as shown in Figure 3 1 It receives a 2-rail input *DataIn* 2-rail outputs are available on Q_1 to Q_4, by the application of the *Data_c* control signal Control signal *Shift_c* is used to shift in the applied data while *Shift_out_c* is used to take data out of the last register Completion of this is acknowledged on the *ShiftOutAck* terminal A transition on *ShiftInAck* indicates that new data bit can be applied at *DataIn*

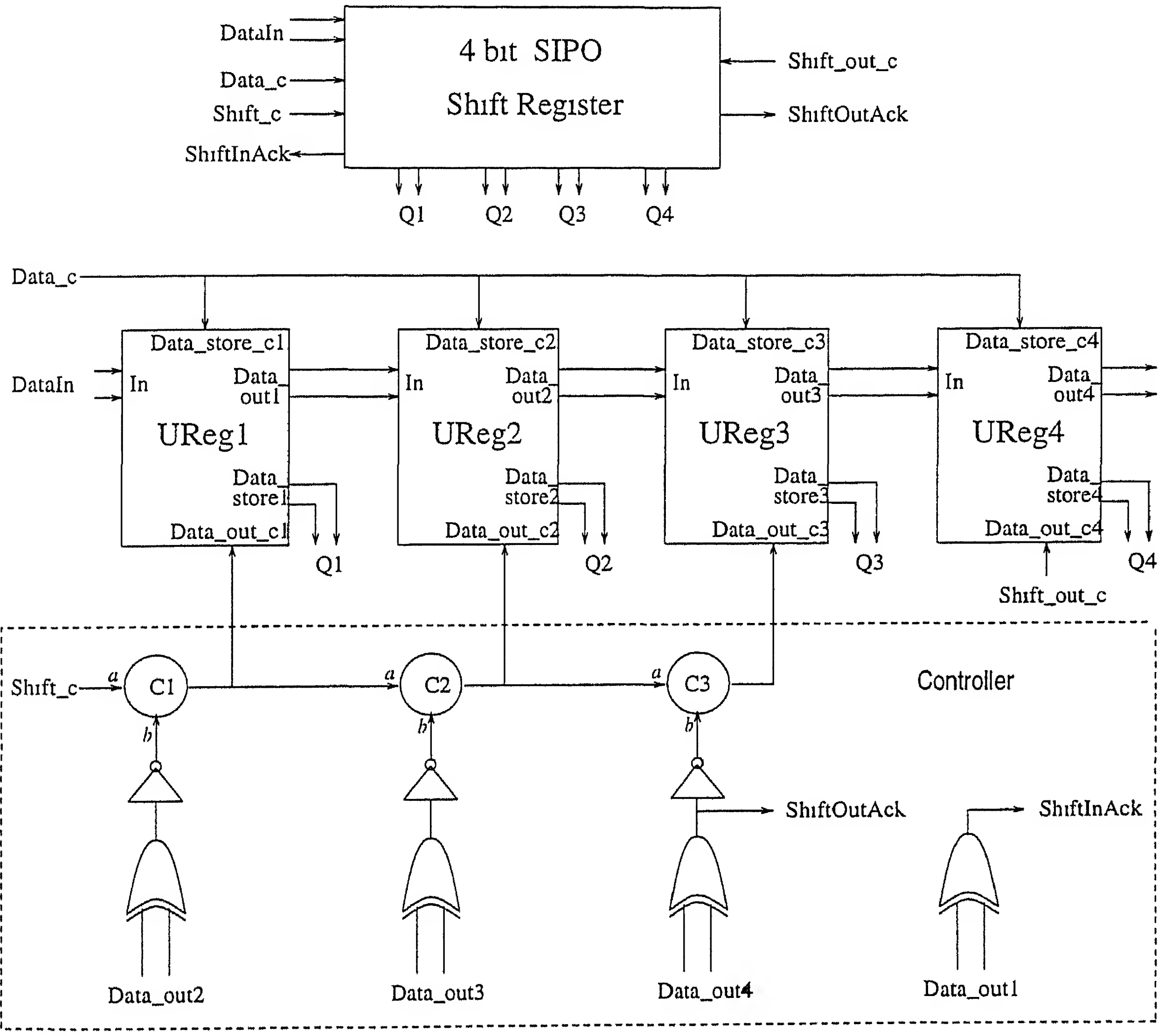


Figure 3 1 Serial In Parallel Out Shift Register

The realisation consists of two distinct blocks as shown in Figure 3 1 The first block has four URegs while the second block is that of the controller controlling the data transactions which takes place between the adjacent URegs In this realisation, on application of a global reset, no data is present in the shift register and the controller is taken to an appropriate state A state in the controller implies transitions present at the inputs of the C element C_1, C_2, C_3 Due to the inverters connected to the C elements, on the a global reset, the *b* input of every C element receives a transition

The application of a transition on *Shift_c* will generate a transition on *Data_out_c*$_1$ This eventually results in transitions on *Data_out_c*$_2$ and *Data_out_c*$_3$ When the UReg1 to UReg3 receive a transition on their *Data_out_c* terminal, they pass the value present on their respective inputs to their *Data_out* terminals Therefore the data applied to *DataIn* of the shift register passes through UReg1, UReg2 and UReg3 to finally reach UReg4 The resulting transitions on *Data_out*$_2$ and *Data_out*$_3$ are used to deposit a transition on each of the b inputs of C_1 and C_2, while the b input of C_3 does not receive any transition A transition on *Data_out*$_1$ is converted into *ShiftInAck* indicating that a new data bit can be applied

The next transition on *Shift_c* generates only *Data_out_c*$_1$ and *Data_out_c*$_2$ and not *Data_out_c*$_3$ The newly applied data bit is therefore shifted to UReg3 In a similar manner, a new data can be stored in UReg2 However, when a new data bit is to be stored in UReg1, it is not necessary to have a transition on *Shift_c* However, if *Shift_c* were to be applied, it would remain unconsumed for the final bit stored in UReg1 This process loads a new data in the shift register The data loaded in the shift register can be made available on the outputs Q_1 to Q_4 by each application of *Data_c* signal

To clear the stored data present in the shift register, *Shift_out_c* signal is used This drives the data contained in UReg4 to its *Data_out*$_4$ terminals This causes a transition to be deposited on the b input of C_3 C_3 has a transition waiting on its a input as a result of application of data bit to UReg4 during the loading phase This will finally result in a transition on *Data_out_c*$_3$ which causes the data stored in UReg3 to be shifted to UReg4 Similarly, data in UReg2 is shifted to UReg3 and that in UReg1 to UReg2 As described earlier, the unconsumed transition on *Shift_c* during the loading phase is used up here At the end of this, no data is contained by UReg1 Thus, during the clear phase, we see an operation similar to that of the shift right operation taking place However, the actual shift right phase will involve new data being applied on the *DataIn* terminals along with transitions on control signals *Shift_c* and *Shift_out_c* The intermediate generation of *ShiftOutAck* enables the application of the next *Shift_out_c* signal to clear one more bit in the shift register Repeated application of *Shift_out_c* will remove all the data loaded initially in the shift register

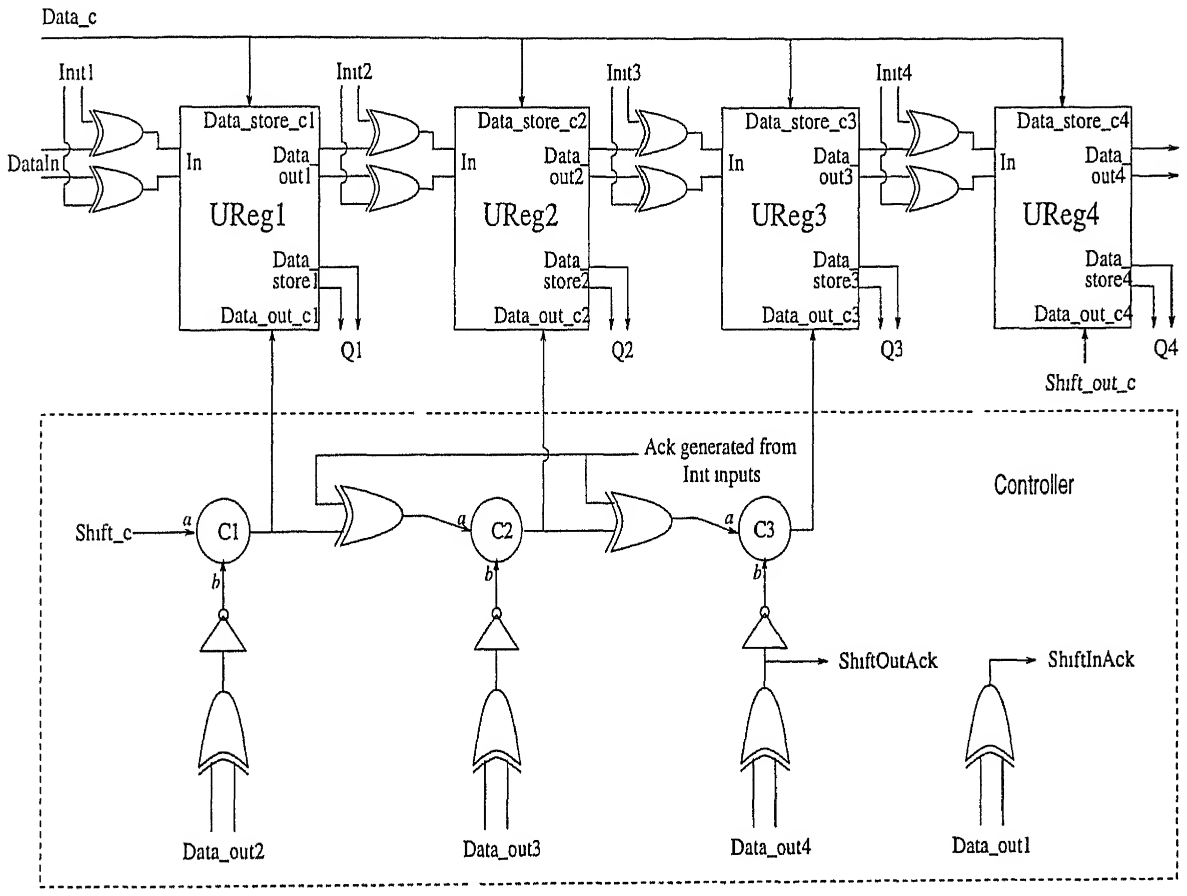


Figure 3 2 Serial In Parallel Out Shift Register Parallel loading

Shifting a new data bit while removing one from UReg4 is possible by simultaneously applying *Shift_c* and *Shift_out_c* along with the application of new data bit on *DataIn* terminals This can be done for both partially, or a completely filled shift register

The above circuit can be modified to enable parallel loading This is shown in Figure 3 2 The controller needs to be modified to result in a state corresponding to that of a fully loaded shift register, corresponding to its being fully loaded This is done by depositing a transitions on the *a* inputs of C_2 and C_3 These transitions can be derived from the initialization input transitions

Other shift register configurations are also possible with relevant changes in the controller and URegs

3.2 Polynomial Serial Parallel Multiplier

Serial Parallel Multiplier implements one of the possible ways of carrying out a multiplication A polynomial multiplication is a multiply without carries [13] It is used in error correction coding

Let d and b be two three bit polynomials, where

$$\begin{aligned} d &= d_2\, x^2 + d_1\, x^1 + d_0\, x^0 \\ b &= b_2\, x^2 + b_1\, x^1 + b_0\, x^0 \end{aligned}$$

where, b_0, b_1, b_2, d_0, d_1, d_2, take values from $\{0,1\}$ The PSPM performs multiplication on two such polynomials Let d be the multiplier, and b be the multiplicand polynomial Let the resultant polynomial be s Then s is given as,

$$s = s_4\, x^4 + s_3\, x^3 + s_2\, x^2 + s_1\, x^1 + s_0\, x^0$$

where, s_0, s_1, s_2, s_3, s_4, s_5, take on values from $\{0,1\}$ and are obtained according to the following Boolean expressions

$$\begin{aligned} s_4 &= d_2\ AND\ b_2 \\ s_3 &= (d_2\ AND\ b_1)\ XOR\ (d_1\ AND\ b_2) \\ s_2 &= (d_2\ AND\ b_0)\ XOR\ (d_1\ AND\ b_1)\ XOR\ (d_0\ AND\ b_2) \\ s_1 &= (d_1\ AND\ b_0)\ XOR\ (d_0\ AND\ b_1) \\ s_0 &= d_0\ AND\ b_0 \end{aligned}$$

The description of the PSPM is organized as follows First the datapath is explained Then a brief idea of the controller is given It is followed by the circuit operation, along with the description of the controller structure Finally, an efficient version of PSPM is given This is obtained by modifying the design presented below

The datapath of a 4-bit Polynomial Serial Parallel Multiplier is shown in the Figure 3 3 It consists of two shift registers, each four bit wide These shift registers

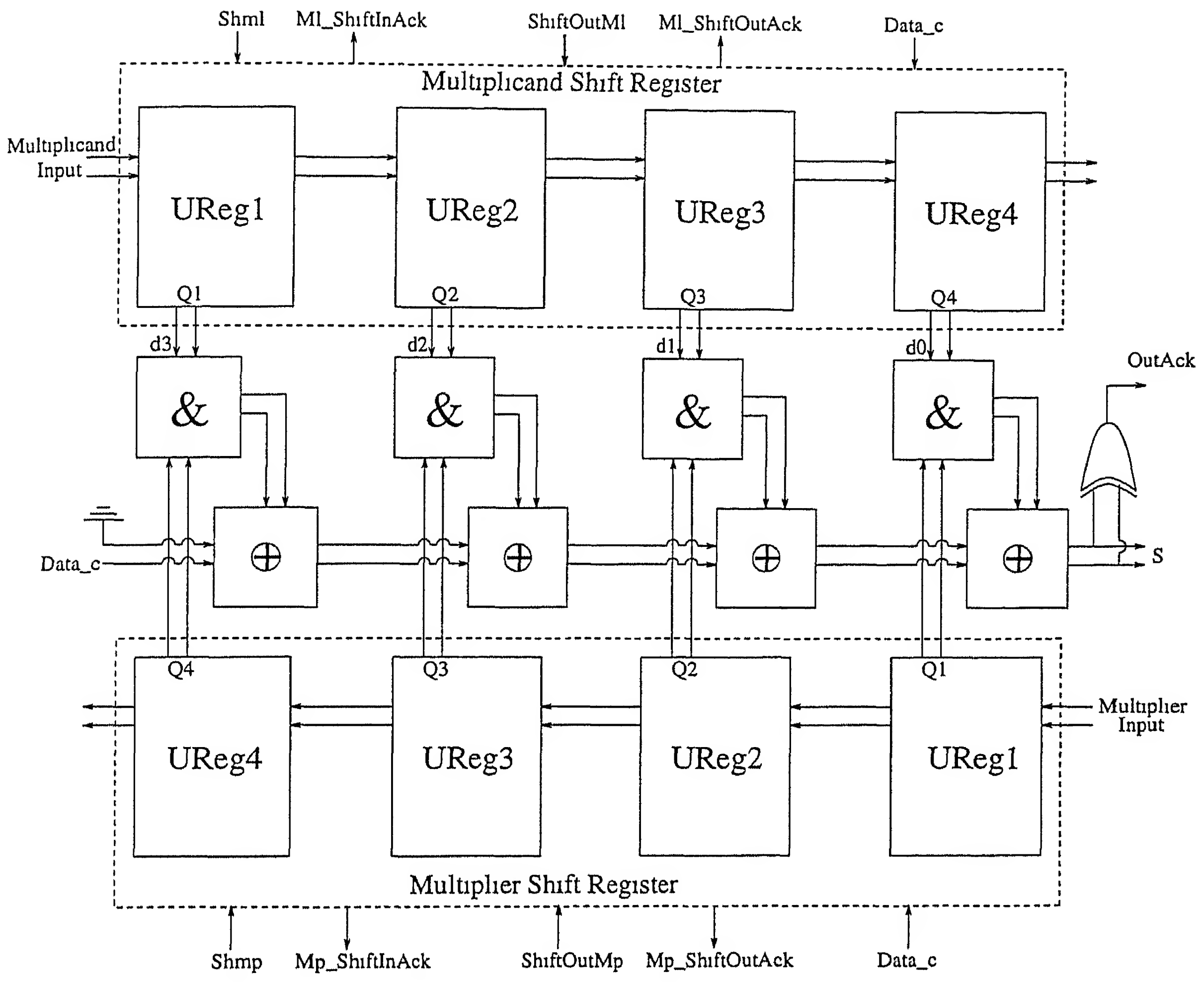


Figure 3 3 PSPM Datapath

store the multiplicand and the multiplier respectively The U gates in the datapath realise the necessary logic functionality to obtain the resultant polynomial All the seven bits of the product are serially available, lsb first, on the output terminals *S* The related control signals for the shift registers are indicated in the Figure 3 3, however the associated controller is not shown for the sake of simplicity

One input of the left most XOR gate needs to be permanently connected to 0 The transition corresponding to the 0 is obtained by tying the corresponding rail to the *Data_c* signal *Data_c* causes both the shift registers to output the stored data corresponding to the multiplicand and multiplier This signal therefore initiates the computation of a single bit of the product polynomial The end of each computation is indicated by a transition on the *OutAck* terminal

The controller shown in Figure 3 4a, employs three loops The execution of the first loop loads the shift registers with the multiplicand and the multiplier The second loop controls the computation of the product polynomial s and the third loop clears the stored data from the shift registers so that new set of data can be applied

To control the iteration count of each of the three loops, the controller employs an Event Counter The Event Counter counts the number of events taking place on its *Inc* input This is a modulo 15 counter to take into account all the iterations of the three loops The count is available on C_1 to C_{15} in decoded form for each of the three loops

These outputs are processed to derive the necessary information for the controller For example, the 2-rail signal *Count3* shows 0 for C_1, C_2 and C_3 and becomes 1 on C_4 It does not respond to any of the other Event Counter outputs While the signal Count11 shows a 0 for outputs C_4 to C_{10} and 1 for C_{11}, not responding to any other Event Counter outputs

Figure 3 4b shows the additional circuitry required to generate control signals for the datapath elements , e g the control signals *Shmp*, *Data_c*, etc *Inc* is an another example of the control signal Similarly acknowledge signals from the datapath are converted into relevant acknowledge signals for the controller using the circuits shown in the Figure 3 4b These are discussed later

Multiplier Operation :

The multiplication operation is initiated in the PSPM with a transition on its *Start* terminal This generates an event on C_1 of the Event Counter, which initiates execution of the first loop in the controller

In this, the multiplicand is loaded in the multiplicand shift register, lsb first, such that Ureg4 contains lsb At the same time, UReg2 to Ureg4 of the multiplier shift register receive 0 The loop is executed thrice, so that three of the four URegs of both the shift registers receive data Each loop iteration increments the count by one Execution of the consecutive iterations is separated by *ShiftInAck* signals issued by the shift registers to the controller to ensure hazardless functioning A transition on the wire w_4 in the controller is used to put the msb of multiplicand in the UReg1 of the corresponding shift register

In the first iteration of the second loop, lsb of the multiplier is stored in UReg1 At the same time, both the registers output their contents to the U gates implementing the logic expressions for s All the bits, excepting the lsb of the multiplier shift register are zero Hence the least significant bit obtained on terminals S will be given by (d_0 AND b_0) After receiving the *OutAck* signal, the multiplier data is shifted left by one bit This puts the lsb of the multiplier in UReg2 In the next iteration, the next significant multiplier bit is applied and the next computation is initiated The process repeats seven times, so that all the seven bits of the product are computed In the last three iterations, a 0 is shifted in the multiplier shift register

The four iterations of the third loop clears the shift registers for the next multiplication

It can be seen that, *Mp_ShiftInAck* signal of the multiplier shift register is generated for every iteration in the first two loops and once in the third loop As this signal is available on a single wire, it needs to be decoded to two different acknowledge signals for the first and second loops, while the occurance of this signal in the third loop should be accounted for separately This is achieved by the two Select blocks and an XOR gate shown in the Figure 3 4b This signal *Mp_ShiftInAck* is produced on *Mp_ShiftInAck1* in response to a transition on wire w_2 of the first loop in the controller It is produced on *Mp_ShiftInAck2* in response to a transition on w_7 while a transition on w_9 produces it on terminals NC which are not connected to any block A similar arrangement exists for *Ml_ShiftInAck* and *Mp_ShiftOutAck*

The total execution time for the PSPM can be reduced by assuming that both the shift registers contain some arbitrary data on global reset The loading of new data for each multiplication operation can then be performed while simultaneously removing the stored data corresponding to the previous multiplication operation

Minor modifications in the datapath involves using an inverter in any of the 2-rail input for all the URegs used in the shift register of Figure 3 3 Also the inverters connected to the b input of all the C elements C_1 to C_3 are removed Inverters are added to the a input of the C elements C_2 and C_3 These inverters in the shift registers ensure initial data after a global reset The modification in the controller of the shift register reflects the modified status of the shift register

The modified controller for the PSPM is as shown in Figure 3 5 and is self explanatory

3.3 Modulo 8 Initializable Up/Down Counter

As the name suggests, the objective of this circuit is to count both in the up and in the down direction The counter is also initializable using the *Clear* and *Init* signals Besides double rail input terminals $Init_0$, $Init_1$, and $Init_2$, it has *Up_count* and *Down_count* control input terminals which set the mode of counting A single rail *Clear* terminal is meant for removing all the transitions corresponding to the current count present in the Counter The completion of this operation is signified by an event on *ClrAck* After global reset or a *Clear*, initialization inputs can be applied on *Init* inputs This can be followed by an event on either the *Up_count* or the *Down_count* input

The Counter is designed exactly as its synchronous counterpart It is then implemented with the A_CTff and A_Tff and the necessary U gates to realise Next State equations as listed in Figure 3 6 Figure 3 6 also shows the implementation of the Counter The *Up_count* and *Down_count* control signal wires are disjunctively combined to derive the *Next_state* control input for all the flipflops The *ClrAck* signal of all the flipflops are used to get a *ClrAck*

The circuit block B receives as its inputs the *Cascade* output of both the A_CTff and A_Tff It also has the *Up_count* and the *Down_count* control inputs In the down count mode, it inverts the input while in the up count mode, the inputs reach the output terminals uncomplemented This block is very easily designed using a U gate and two XOR gates

Every time the flipflops change state, after receiving the *Next_state* input, the present state is available on the corresponding *Cascade* terminals These state bit values are then processed to derive the Next State T input, for each of the flipflops This is done using the B block and the AND gate implemented using a U gate

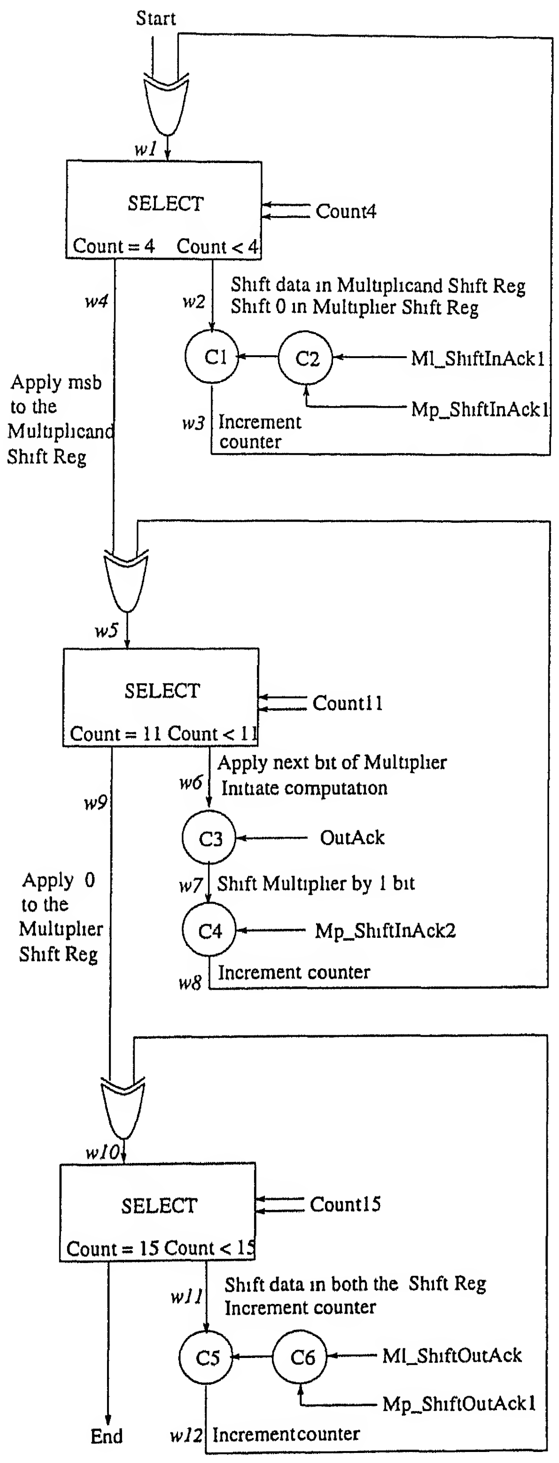
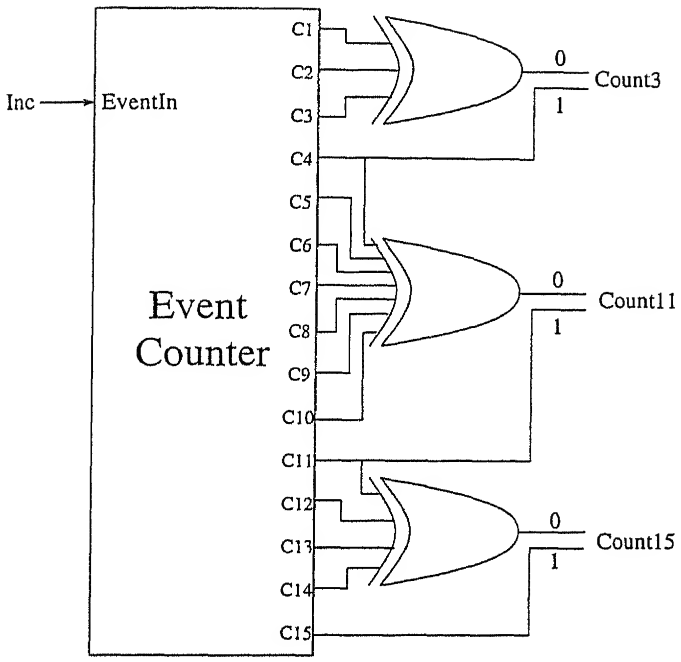
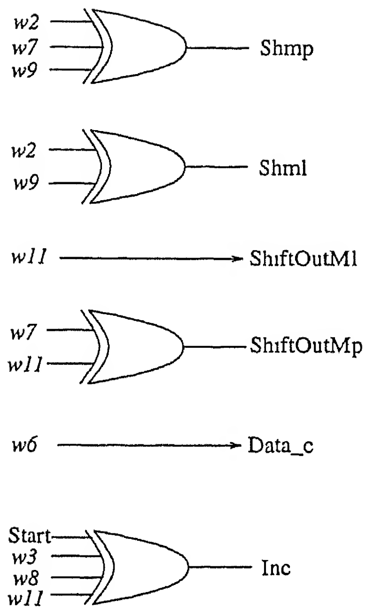


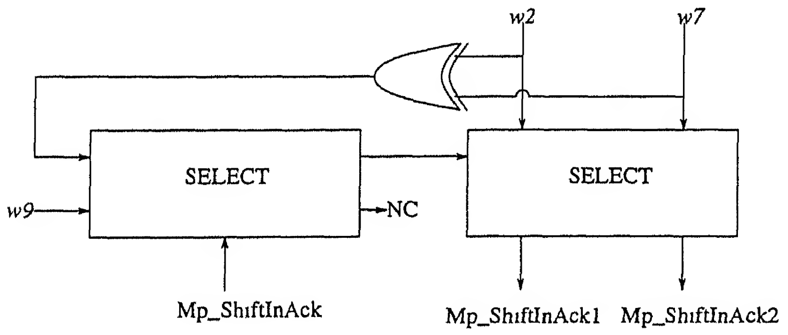
Figure 3 4a PSPM Controller



Circuit to generate different count signals



Control signal for datapath generated from controller



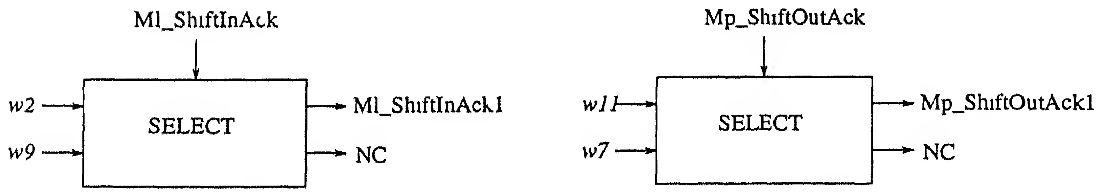
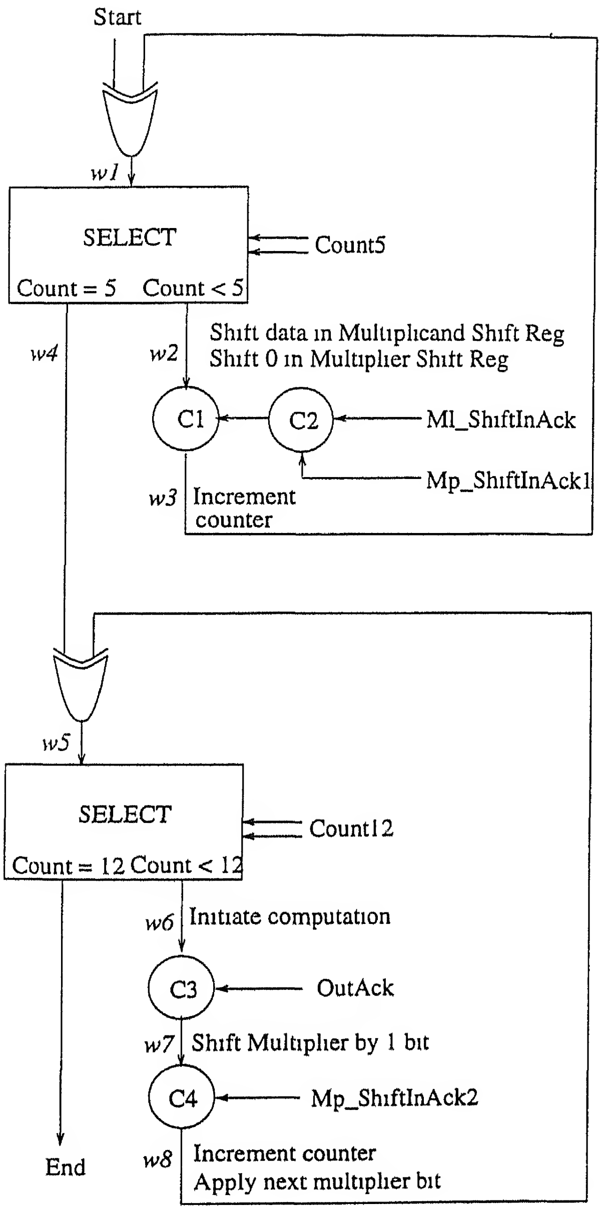
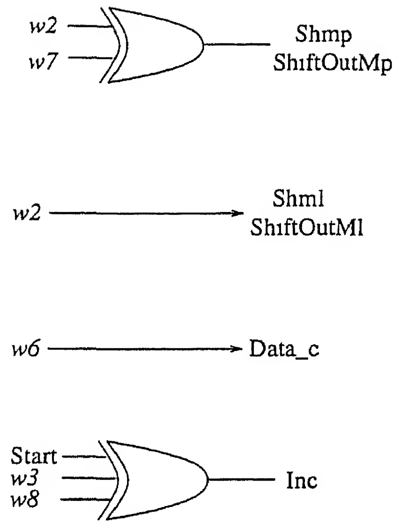
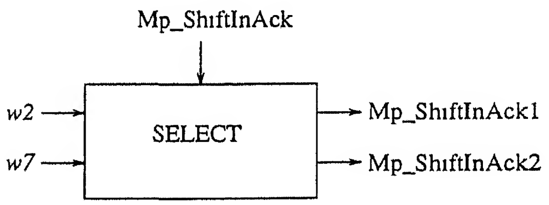


Figure 3 4b PSPM Controller



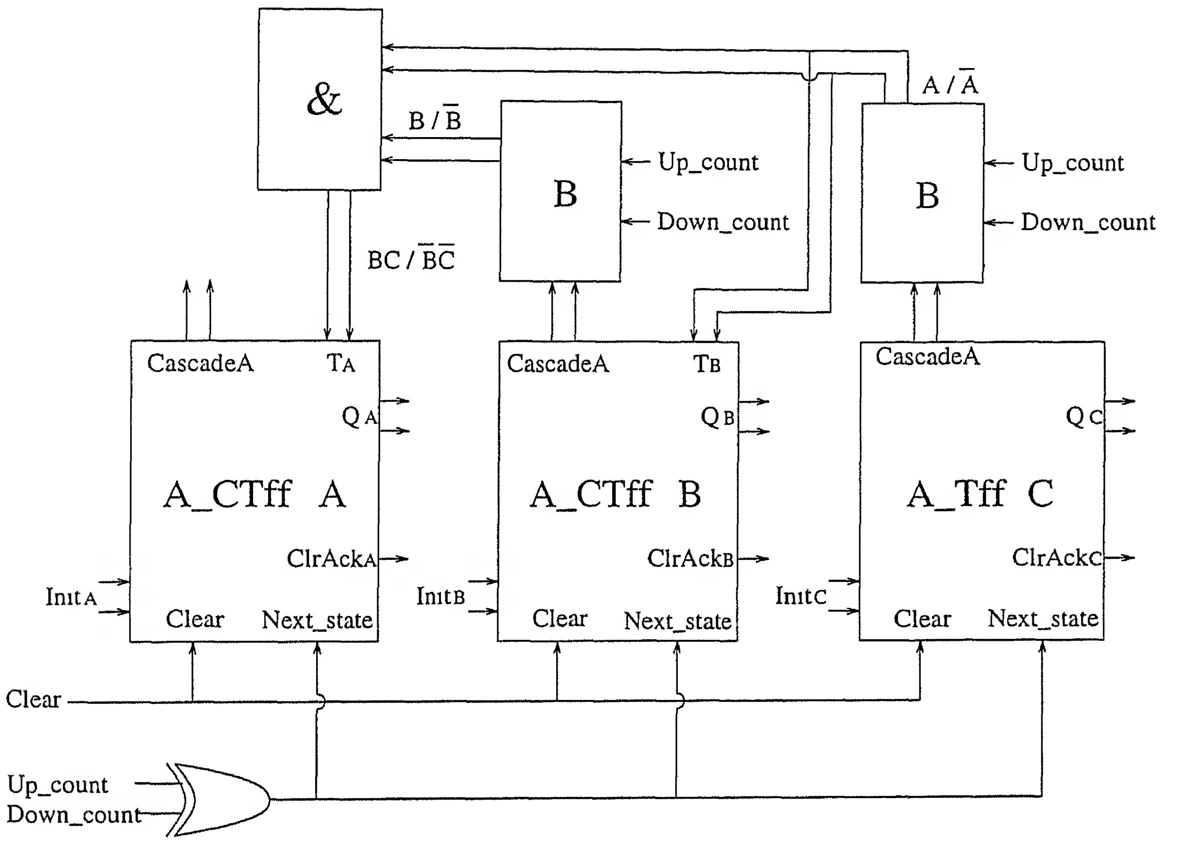


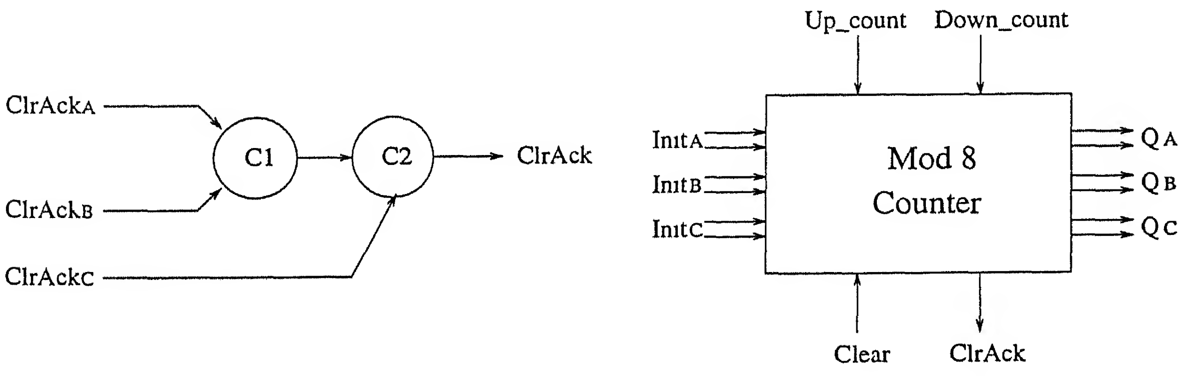
Control signal for datapath generated from controller



Acknowledge signals for the controller are generated from the datapath acknowledge signals

Figure 3 5 PSPM Controller (Improved Version)





Up Counting mode $T_c = 1,\ T_b = C,\ T_a = BC$

Down Counting mode $T_c = 1,\ T_b = \overline{B},\ T_a = \overline{B}\overline{C}$

Next State Equations

Figure 3 6 Modulo 8 Up Down Counter

Chapter 4

Interconnections

4.1 Introduction

Different resources like registers, datapath elements and input and output ports need to be interconnected to realise any digital system The interconnections can be carried out in several ways, out of which, the point to point interconnection and the bus topology have been widely used in the synchronous systems In this chapter, we explore the possibility of employing similar interconnection structures for realising digital systems using asynchronous elements In the point to point interconnection scheme, there is a direct connection from output of one resource to the input of another resource which needs it However, due to resource sharing, it becomes necessary at different instants of time, to connect one resource output out of several resources to the input of another resource While in the bus topology, the output of each resource is transferred to a common resource shared by all of them, usually the bus The output is then routed to the desired destination

4.2 Point to Point Interconnections

Assume that the data in the register is to be transferred to only a single destination Then a direct connection is made between the two However, if the data is to be transferred to more than one resource, then a direct connection is not possible To realise this task, the output of the source register needs to be routed to the appropriate destination This can be realised using a A_Demux Functional representation of the A_Demux is as shown in Figure 4 1a *In* is the two rail input C_1 to C_n are the n single rail control signals A_Demux has n 2-rail outputs O_1 to O_n The application of one of the control signals C_i transfers the input data to the corresponding output O_i The circuit realisation for $n = 4$ is as shown in the Figure 4 1a

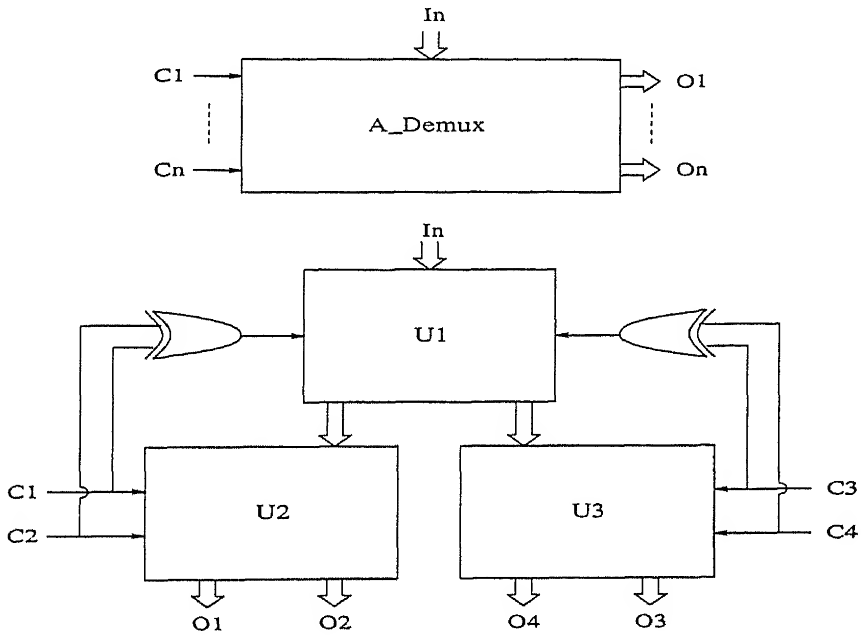


Figure 4 1a A_Demux

An application of *Data_out* and one of the control signals C_i to the A_Demux will transfer the data in REG1 to O_i as shown in 4 1b It can be seen that the realisation for $n = 1$ is a Select block

4.3 Bus Structures

The bus interconnection scheme can be studied in the following general setting Assume the following resources are given.

1 A register file of n registers, each m bit wide

2 pi input ports and po output ports, each m bit wide

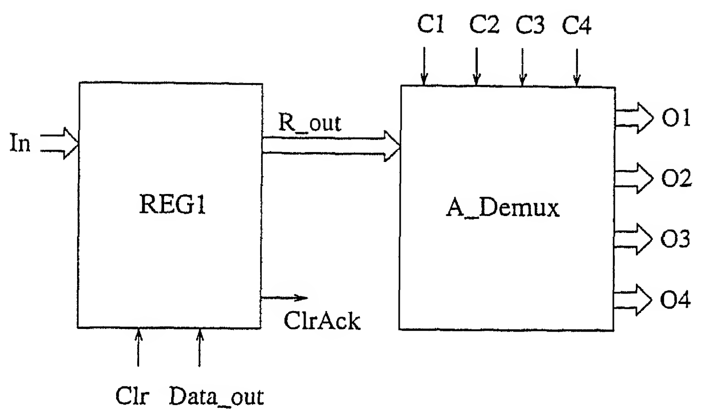


Figure 4 1b Data transfer using A_Demux

3 An m bit wide bus which is nothing but m 2-rail wires i e $2m$ wires

The bus interconnection should allow the following classes of data transfer for the resources given above

1 Register to register

2 Register to output port

3 Input port to register

In the descriptions that follows, the register to register data transfer will be indicated by $R_i \longrightarrow R_j$, where i indicates the source and j the destination The register to port data transfer is indicated by $R_i \longrightarrow P_j$ and the remaining data transfer is indicated by $P_i \longrightarrow R_j$

In the design methodology adopted, any register communicates data with the datapath elements in the same way as it communicates it with the ports Hence, datapath elements receive data through $R_i \longrightarrow P_j$ and their output is routed to registers through $P_i \longrightarrow R_j$

The typical bus structure is as shown in Figure 4 2 It consists of n registers, p_0 output ports and p_i input ports connected together using the bus The data transfers between them are coordinated by the local bus controller The local bus controller receives the control signals from the main controller CON1 For each data transfer carried out on the bus, an event on the *FinalAck* is generated, signifying the completion of the data transfer This signal is sent to the CON1

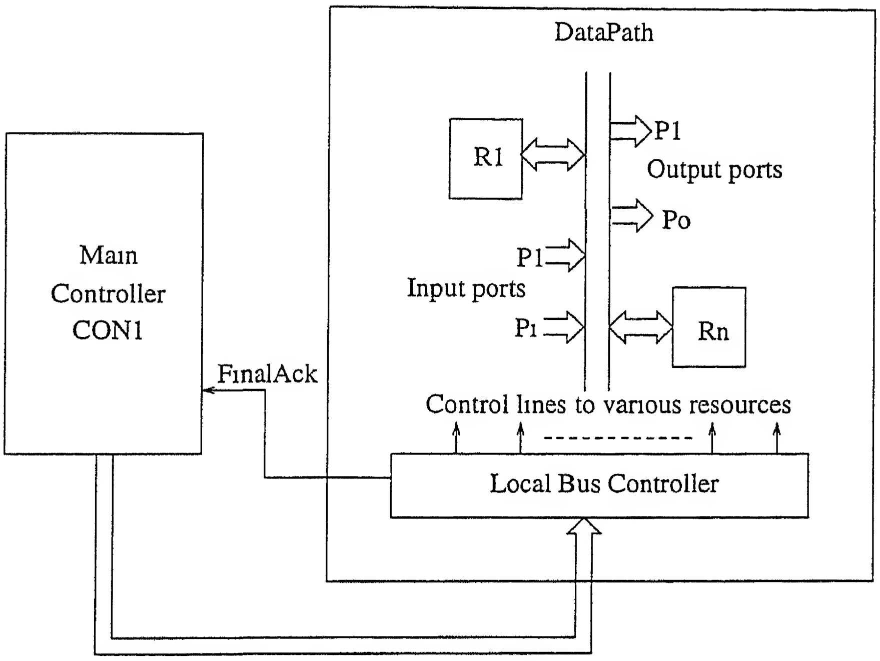


Figure 4 2 Block diagram of a bus structure

We assume that the main controller CON1 provides the following control signals to the local bus controller

1 For the n registers

   (a) n source control wires, WS_1 to WS_n

   (b) n destination control wires, WD_1 to WD_n

2 po register to port transfer control signals, WP_1 to WP_o

3 A port to register transfer control signal, $P2r\_trf$

Furthermore, it is assumed that the following events are available to the bus structure for different modes of data transfer

$R_i \longrightarrow R_j$ A transition each on WS_i and WD_j

$R_i \longrightarrow P_j$ A transition each on WS_i and WP_j

$P_i \longrightarrow R_j$ A transition on WD_j and on $P2r\_trf$

All the above assumptions are made in order to isolate the design of the bus structure from the details of the main controller CON1

4.3.1 Basic Idea

All the three approaches to be discussed use the same underlying basic idea as given below Two distinct phases constitute a typical data transfer, Data clearance and Data transfer to the destination

Data clearance This phase is carried out only when the destination is a register In this, the data in R_j is cleared and a *ClrAck* is generated to signify the completion of the operation

Data transfer The bus receives data from all the input ports and from the outputs of all the registers All these 2-rail signals corresponding to a bit position in an m bit wide bus are merged into a single 2-rail signal through an Merger element The output of the Merger could either be transferred directly to the bus or transferred after necessary processing The bus transfers data to all the registers and output ports The signal on the bus is routed to the desired destination using a 2-rail Data Decoder For every data transfer, a completion signal called *FinalAck* is generated by the bus This serves as an acknowledge signal to the main controller CON1

$R_i \longrightarrow R_j$ This data transfer involves clearing the previous data stored in Rj This is then followed by transfer of data from R_i to R_j

$P_i \longrightarrow R_j$ In this, we assume that the data is already available at the input port The destination register is first cleared as above and then the data present in port P_i is routed to the R_j

$R_i \longrightarrow P_j$ In this, the data in R_i is transferred to the bus and is then routed to the P_j

The rest of the chapter describes three approaches to realise the bus structure Each approach is described as follows First the datapath and the local bus controller are described The operation in the three modes of data transfer is illustrated Circuit complexity in terms of the number of C elements and XOR gates and the delay in implementing a data transfer $R_i \longrightarrow R_j$ is given Finally, the comparison of the three approaches in terms of the circuit complexity and delay is made The delay constraints for the Data Decoder are specified and an alternative implementation of the Basic Data Decoder module is given which can be used in case the delay constraint can not be satisfied easily We also present the implementation of a $P_i \longrightarrow P_j$ data transfer that is needed for transfer of data directly between datapath elements

All the bus structures are implemented for 1 bit For m bits, m instances of the datapath should be used

4.3.2 Approach 1

The datapath needed for this approach is as shown in Figure 4 3a The datapath consists of a 2-rail merger, a Data Decoder, n registers and a 1-rail merger to generate the FinalAck signal

The inputs to the 2-rail merger are the outputs of the registers and the input ports These inputs are combined such that, a transition on any one of them is routed to the output Only one input gets routed to the output at any instant The merger is realised as shown in Figure 4 3b The output of this merger is connected to the bus The input of the Data Decoder is also connected to the bus The Data Decoder receives n *Reg_out* and p_0 *Port_out* control signal from the local bus controller It has $(n + p_0)$ 2-rail outputs Depending on a transition on one of the *Reg_out* or *Port_out* control signals, the input data is routed to the appropriate port or the register Each time an input data is transferred by the Data Decoder, an acknowledge is generated on one of its $(n + p_0)$ *DdrAck* and *PortDdrAck* outputs All these acknowledges are disjunctively combined to derive the *FinalAck* signal by a 1-rail merger The register used in this approach is REG1 described earlier in chapter 2

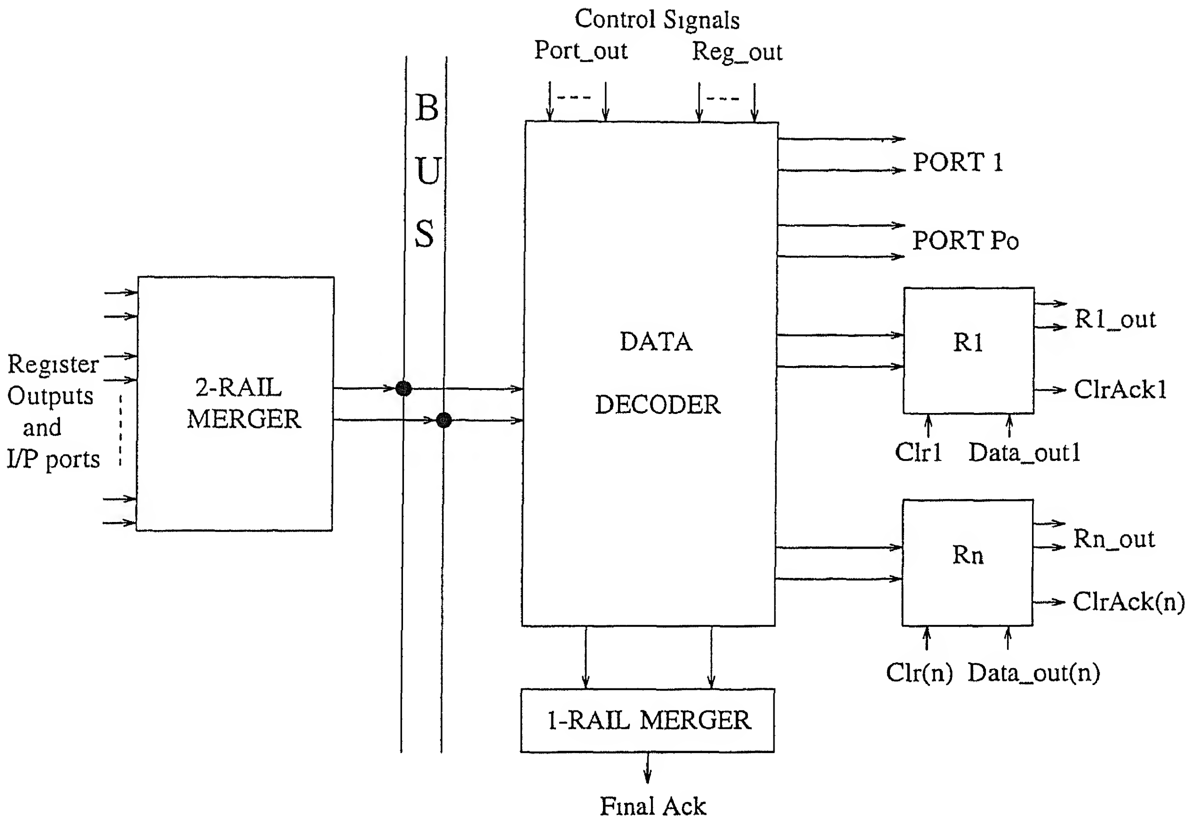
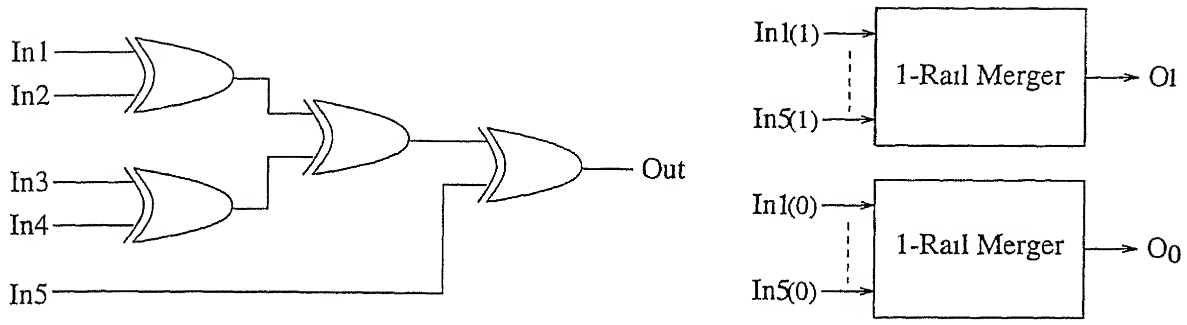


Figure 4 3a Bus approach1 Datapath

The Basic Data Decoder module, instances of which are used to realise the Data Decoder, is shown in Figure 4 3c It has a 2-rail input *In* and a control signal, *Control* It also receives feedback from all other Basic Decoder modules used in the realisation of the Data Decoder It outputs on the *Out* terminals An XOR gate connected to the *Out* terminals generate an *DdrAck* signal, which signifies the completion of the data transfer from the input of Basic Data Decoder module to its output C elements C_1, C_2 and the XOR gate connected to their y inputs constitute a Select block Every time a data bit is delivered by the bus, it is received by the x input of either C_1 or C_2 It can be inferred from the functionality of the Select module that, a transition on the *Control* input terminal will produce the transition on the *Out* terminals This transition corresponds to the transition representing the input data An *Ack* signal signifies the completion of the transfer of the input data to the output terminals Only one of the $(n + p_0)$ Basic Data Decoder modules in



For a 2-rail merger combining n signals
Number of XOR gates = 2(n-1)
Delay = $\log_2(n)$ * XOR gate delay

Figure 4 3b 1-rail and 2-rail merger for 5 inputs

the Data Decoder block transfers the input data to its output terminals However, the remaining Basic Data Decoder modules receive the same input transition These extraneous transitions need to be canceled The transition cancellation is achieved by feeding back the output transition as input to the remaining Basic Data Decoder modules

The local bus controller is shown in Figure 4 4 It consists of n C elements C_1 to C_n One input of C_i is connected to the destination control wire WD_i, which is also connected to the *Clr* input of the Register R_i The other input receives the *ClrAck* output of R_i The output of C_i is connected to the *Reg_out*$_i$ control signal of the Data Decoder The other inputs to the local bus controller, from the main controller, CON1, are also connected to the different modules in the datapath block as shown in the figure

Circuit Operation

- $R_i \longrightarrow R_j$ In this data transfer, transition on WS_i is applied to the *Data_out* control signal of R_i and that on WD_j is applied to the *Clr* of R_j This activates two concurrent processes, in one of which, the data stored in R_i is placed on its *R_out* terminals It therefore, gets transferred to the Data Decoder In the other process, the data in R_j is cleared, and $ClrAck_j$ is generated This

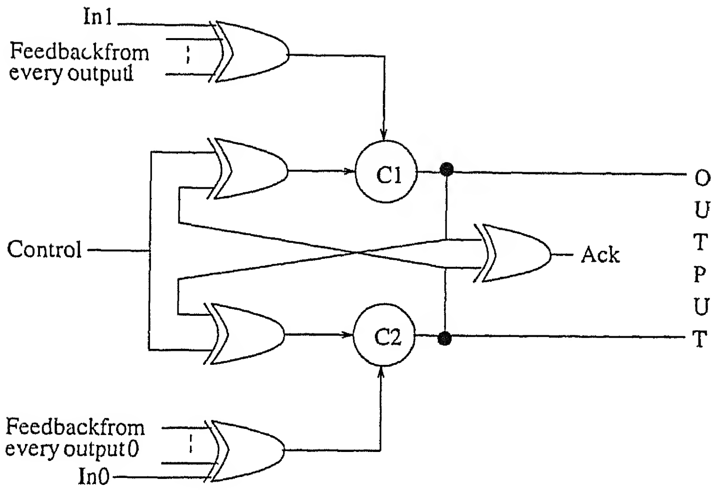


Figure 4 3c Basic Data Decoder module

is applied to C_j whose other input receives a transition from WD_j The corresponding output transition on C_j routes the data transferred to the Data Decoder by R_i to R_j

- $R_i \longrightarrow P_j$ The transition on WS_i is applied to *Data_out* of R_i and that on WP_j is applied to the *Port_out* control signal of the Data Decoder This transfers the data stored in R_i to P_j

- $P_i \longrightarrow R_j$ The data available on P_i gets transferred to the Data Decoder A transition on WD_j clears R_j Subsequently, the *Reg_out* control signal for the Data Decoder is generated This routes the data in the Data Decoder to R_j

▮ *Delay Constraint*

The delay in the $(n + P_0 - 1)$ XOR gates at each input of every Basic Data Decoder module in the Data Decoder is $log_2(n + P_0 - 1)$ times the delay in a XOR gate It can be seen that, this delay is equal to the delay in the 1-rail merger used to generate the *FinalAck* signal For the block consisting of the data decoder module and the 1-rail merger, the XOR gate delays are assumed to be bounded Therefore, this block can be designed such that, by the time, the *FinalAck* is generated, all the extraneous transitions in the Data Decoder are canceled

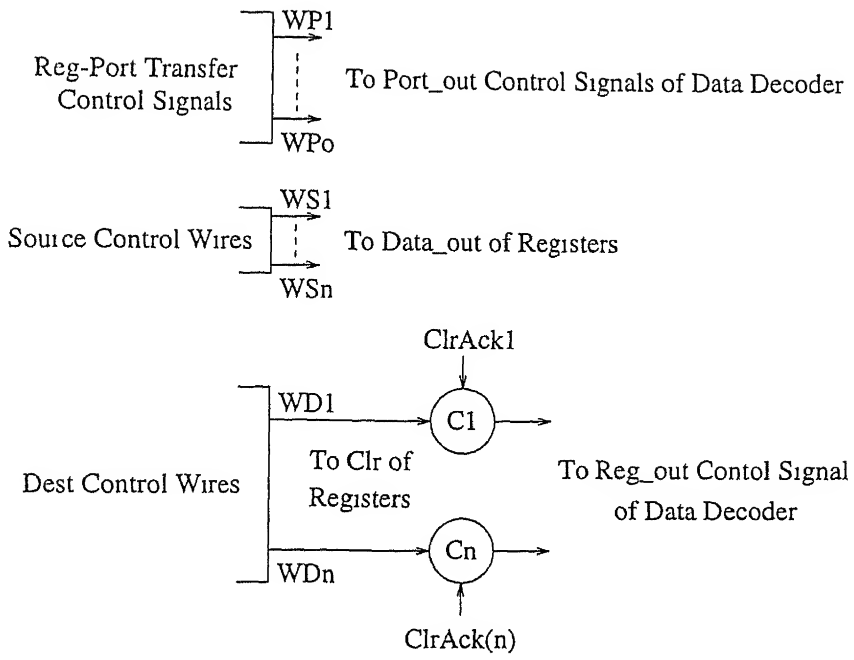


Figure 4 4 Bus Approach 1 Local bus controller

C Element	XOR Gates
$[m(7n+2p_o+1)]-1$	$m[2n^2+2p_o^2+13n+2p_i+2p_o-3]$

Table 4 1 Bus approach 1 Circuit complexity

■ *Complexity*

The circuit complexity is expressed in terms of the number of C elements and XOR gates needed to implement the bus structure It is calculated using the individual circuit complexities of the sub-blocks used The circuit complexity for this approach is as given in Table 4 1

The delays involved in a data transfer are described using a directed acyclic graph $\{V, E\}$, where, V is a set of vertices and E is a set of edges Each vertex represents the completion of an operation in the data transfer The temporal correlation between successive events i and j is represented by a directed edge from V_i to V_j The delay associated with an edge can be expressed in terms of the number of C elements and XOR gates used in generating the successor event We separate the contiibution to the two elements by labeling an edge as $d1/d2$, where, $d1$ is the total delay due to the C elements and $d2$ is that due to the XOR gates The delays involved in $R_i \longrightarrow R_j$ for this approach are as shown in Figure 4 5

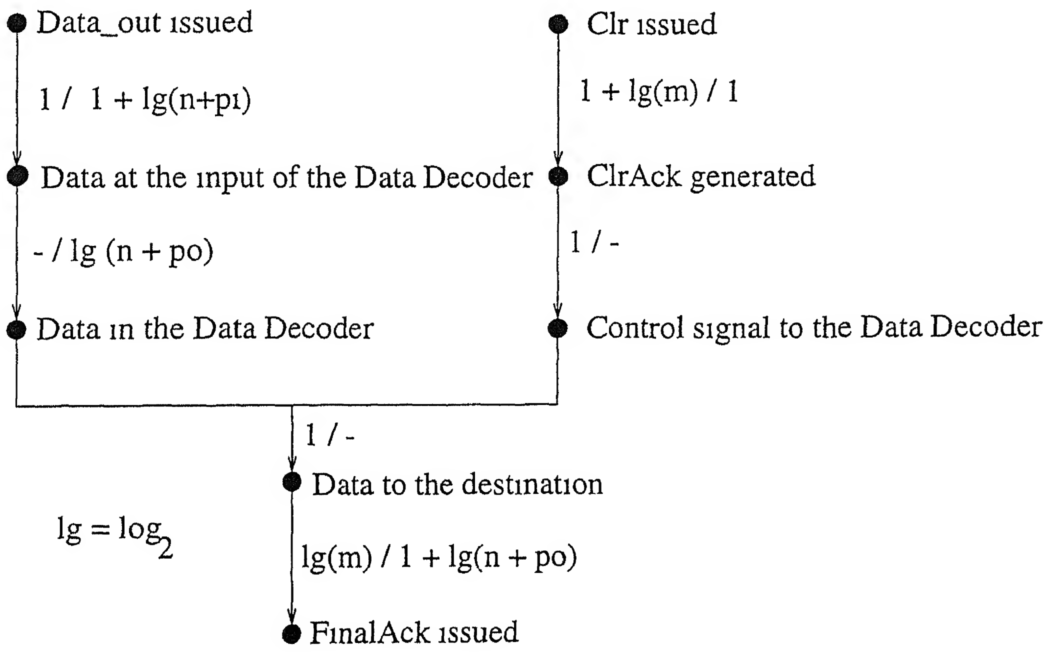


Figure 4 5 Bus Approach 1 Delay in $R_i \longrightarrow R_j$

The major drawback of this approach is the higher circuit complexity of the Data Decoder The number of XOR gates required is proportional to $n^2 + p_o^2$ It can be reduced with an alternative arrangement to derive feedback for each of the Basic Data Decoder modules as shown in Figure 4 6 Here, feedback for each of the $(n + p_0)$ Basic Data Decoder modules is derived using a common network of XOR gates The Basic Data Decoder module uses 2-input XOR gate instead of $(n + p_0)$ input XOR gate The input signal $f_i(1)$, in the Basic Data Decoder module i receives a transition when any of the Basic Data Decoder module transfers the data bit 1 The upper limit on the complexity of feedback processing circuit is $Nlog_2N$, where $N = n + po$ The modified circuit complexity and the delay in $R_i \longrightarrow R_j$ are shown in Table 4 2 and Figure 4 7

In the next approach, a way to reduce the circuit complexity further, is shown

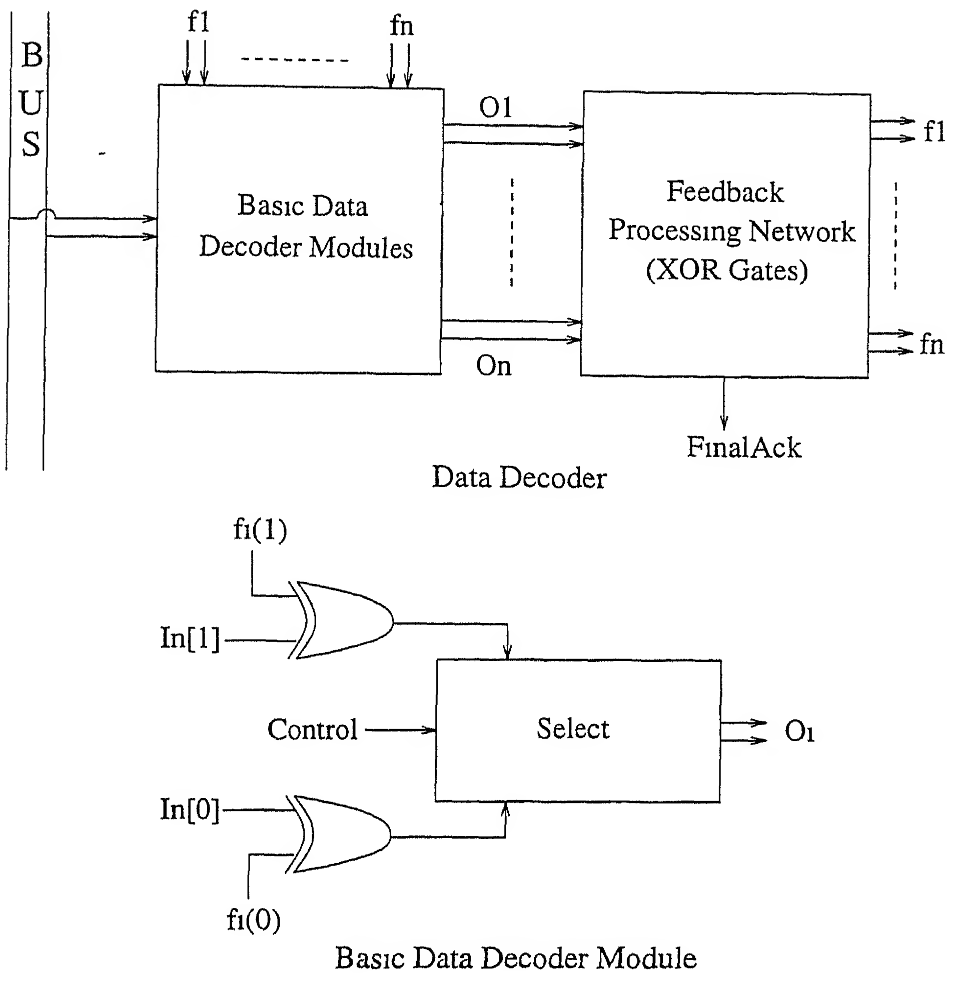

BUS
f1
fn
Basic Data Decoder Modules
O1
On
Feedback Processing Network (XOR Gates)
f1
fn
FinalAck
Data Decoder
fi(1)
In[1]
Control
Select
Oi
In[0]
fi(0)
Basic Data Decoder Module

Figure 4 6 Bus Approach 1 Modified decoder

4.3.3 Approach 2

The cause of the higher circuit complexity of the Data Decoder of approach 1 is the large number of XOR gates used to cancel the extraneous transitions in the Data Decoder In this approach, the circuit complexity of the Data Decoder is reduced by using a 2 input XOR gate instead of a $(n + P_0)$ input XOR gate in the Basic Data Decoder module

In a single data transfer, the Data Decoder used in this approach receives, at its input, the data to be transferred, twice The first occurence of the data is used to route it to the appropriate destination and the second occurence of the data is used to cancel the extraneous transitions in it The application of the data to the

C Element	XOR Gates
$m[7n + 2p_o + 1] - 1$	$m[11n + 2p_i - 2] + [2(n + p_o)(1 + log_2(n + p_o))] + 1$

Table 4 2 Bus approach 1 Circuit complexity with the modified decoder

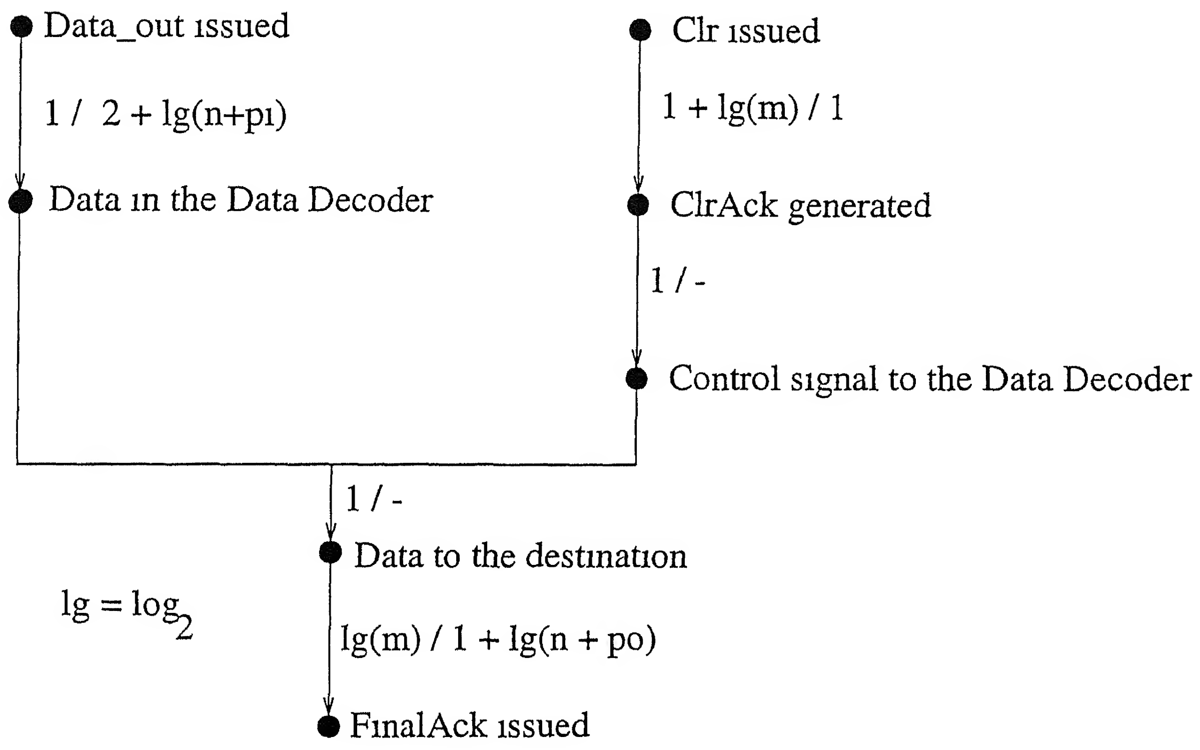


Figure 4 7 Bus Approach 1 Delay in $R_i \longrightarrow R_j$ for the modified decoder

decoder twice in a single data transfer is made possible by the use of a Pass and Store circuit

The datapath to realise this approach is as shown in Figure 4 8a Compared to the approach 1, here the 1-rail merger is replaced by a Final Ack Generator

The Pass and Store block receives the output of a 2-rail merger It has two control signals *Pass_cnt* and *Pass_store_cnt*, and a 2-rail output which is connected to the bus The *Pass_store_cnt* control signal places the input data on the output terminals In the process, it also retains the data While the *Pass_cnt* which is subsequently applied, is used to shift the stored data to its output, after which no data exists in the circuit block This circuit is very easily realised using a UReg and

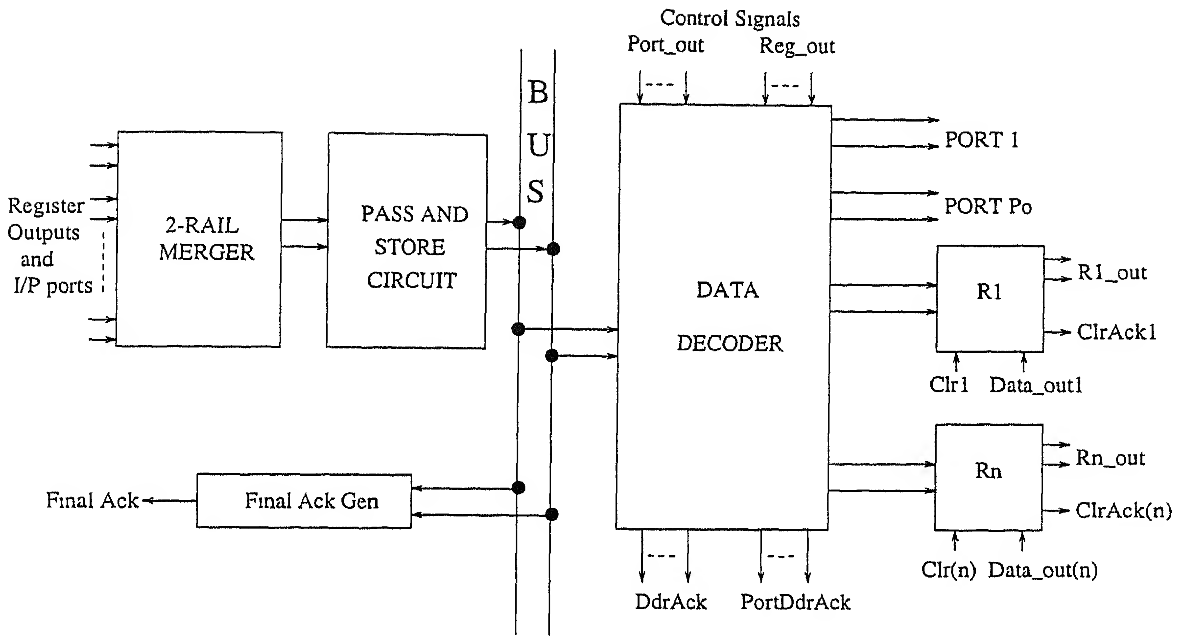


Figure 4 8a Bus Approach 2 Datapath

2 XOR gates as shown in Figure 4 8b

The Basic Data Decoder module is as shown in Figure 4 8c The C elements C_1, C_2 and the XOR gates connected to their y inputs constitute a generalized C element In every data transfer, the first application of data at one of the inputs is transferred to the corresponding C element in the Generalized C element A transition on the *Control* input then transfers the data bit to the OUT terminals Also, the feedback from the output of the C element through the XOR gate places the transition on OUT back into the C element As seen from the structure of the Data Decoder, since the 2-rail bus is connected as an input to all the Basic Data Decoder modules, transition corresponding to the data is consumed only by that Basic Data Decoder module B_i, which receives the Control signal, whereas it is retained by the remaining modules The second application of the same data, therefore cancels these retained transitions as well as the output transition fed back to B_i

The Final Ack Generator receives data from the bus at its input and produces an event on the *FinalAck* signifying the completion of a single data transfer operation

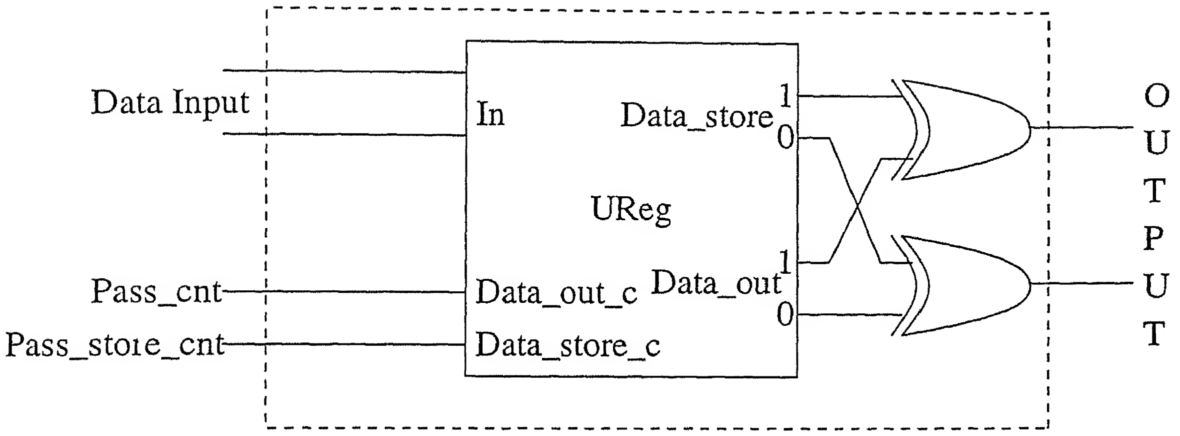


Figure 4 8b Bus Approach 2 Pass and Store Circuit

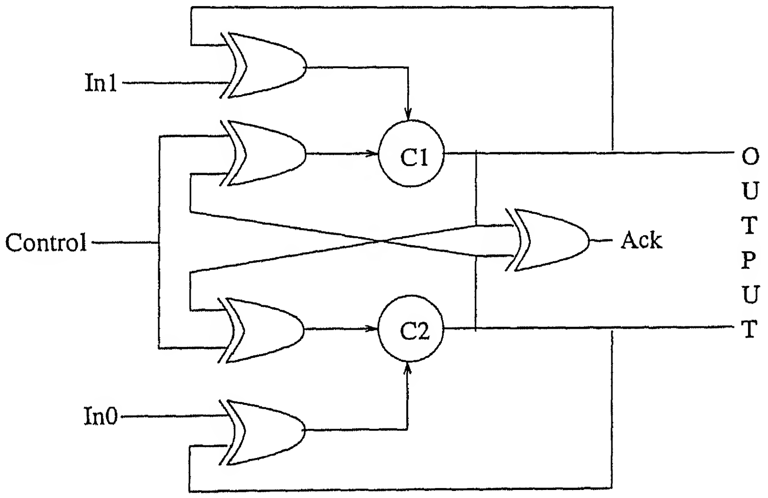


Figure 4 8c Bus Approach 2 Basic Data Decoder Module

It can be easily seen that the *FinalAck* will be generated after the data is transferred to the destination Two realisations of this circuit are shown in Figure 4 8d In the first realisation, the x input of the C element will initially have a transition due to the global reset The first application of the new data cancels this transition Next, a transition on the y input due to *Pass_cnt* will create an output transition after the second application of the new data This generates the *FinalAck* It also takes the Final Ack Generator to the state it was, before the application of new data The second realisation is based on the Toggle element and is the delay insensitive version of the first

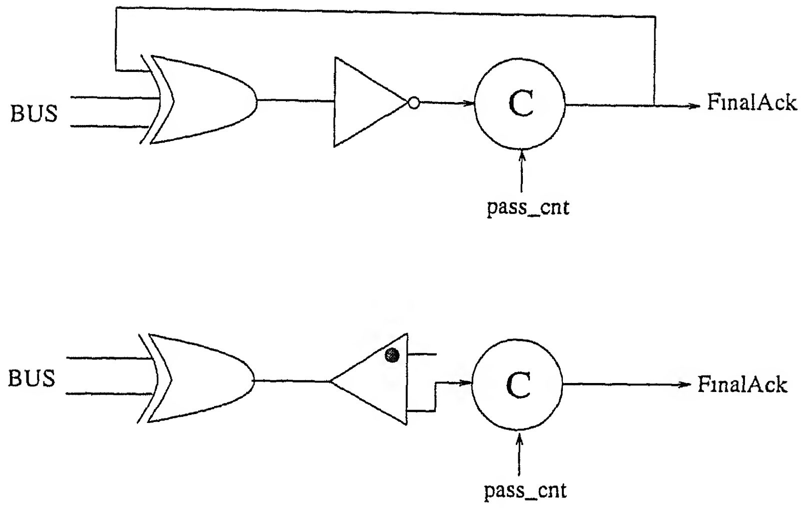


Figure 4 8d Bus Approach 2 Final Ack Generator

In the present approach, the data transfer from the source to destination in all the three modes, is very similar to that in the first approach However, due to changes in the Data Decoder and the addition of a Pass and Store circuit, there are some differences in the actual implementation There are two implementations of the controller for this approach These arise, from the way in which the control signals *Pass_store_cnt* and *Pass_cnt* are generated

▮ *Local Bus Controller 1*

In this controller the control signals *Pass_cnt* and *Pass_store_cnt* are generated locally for each of the m bits as shown in the figure 4 9 The signal *Pass_store_cnt* is generated by disjunctively combining the *ClrAck* signals from the registers and the WP_1 to WP_o wires While the *Pass_cnt* is generated by disjunctively combining all the *DdrAck* and *PortDdrAck* signals of the data decoder Also, as compared to the local bus controller of approach 1, it does not use the n C elements C_1 to C_n Controller of approach 1 uses them to ensure that, in a data transfer to R_j, the data is not transferred to it unless the previous data in it is cleared In this controller the same restriction is satisfied by generating the *Pass_store_cnt* only after the *ClrAck* of R_j is generated

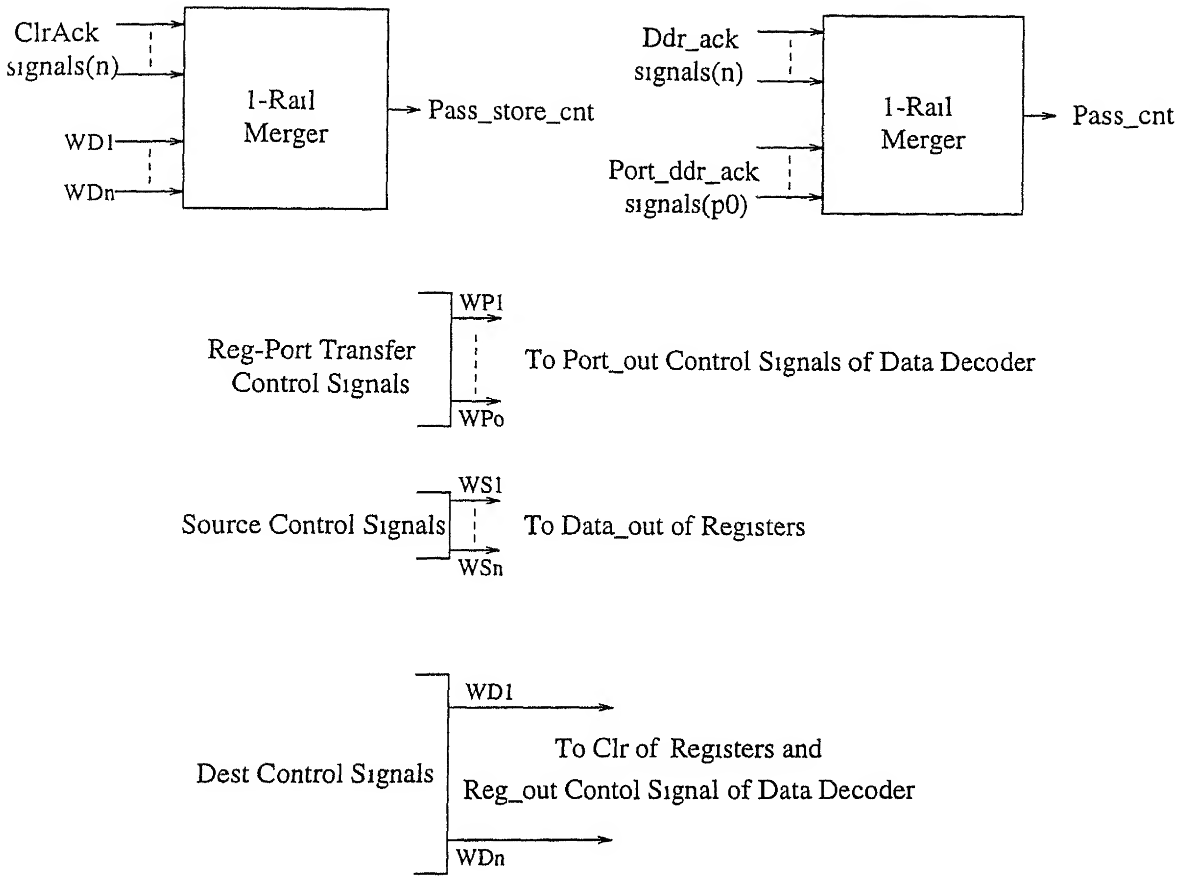


Figure 4 9 Bus Approach 2 Local Bus Controller 1

■ *Local Bus Controller 2*

Instead of generating the *Pass_cnt* and *Pass_store_cnt* locally for each of the m bits using the *ClrAck* and the Data Decoder acknowledge signals, these signals are passed to the local bus controller These signals are processed by the local bus controller to generate the *Pass_cnt* and *Pass_store_cnt* signals as shown in Figure 4 10

■ *Complexity*

The circuit complexity of the datapath and both the controllers is as shown in the Table 4 3 The delays in $R_i \longrightarrow R_j$ for both the controller realisations is represented in the graphs of Figure 4 11 and Figure 4 12

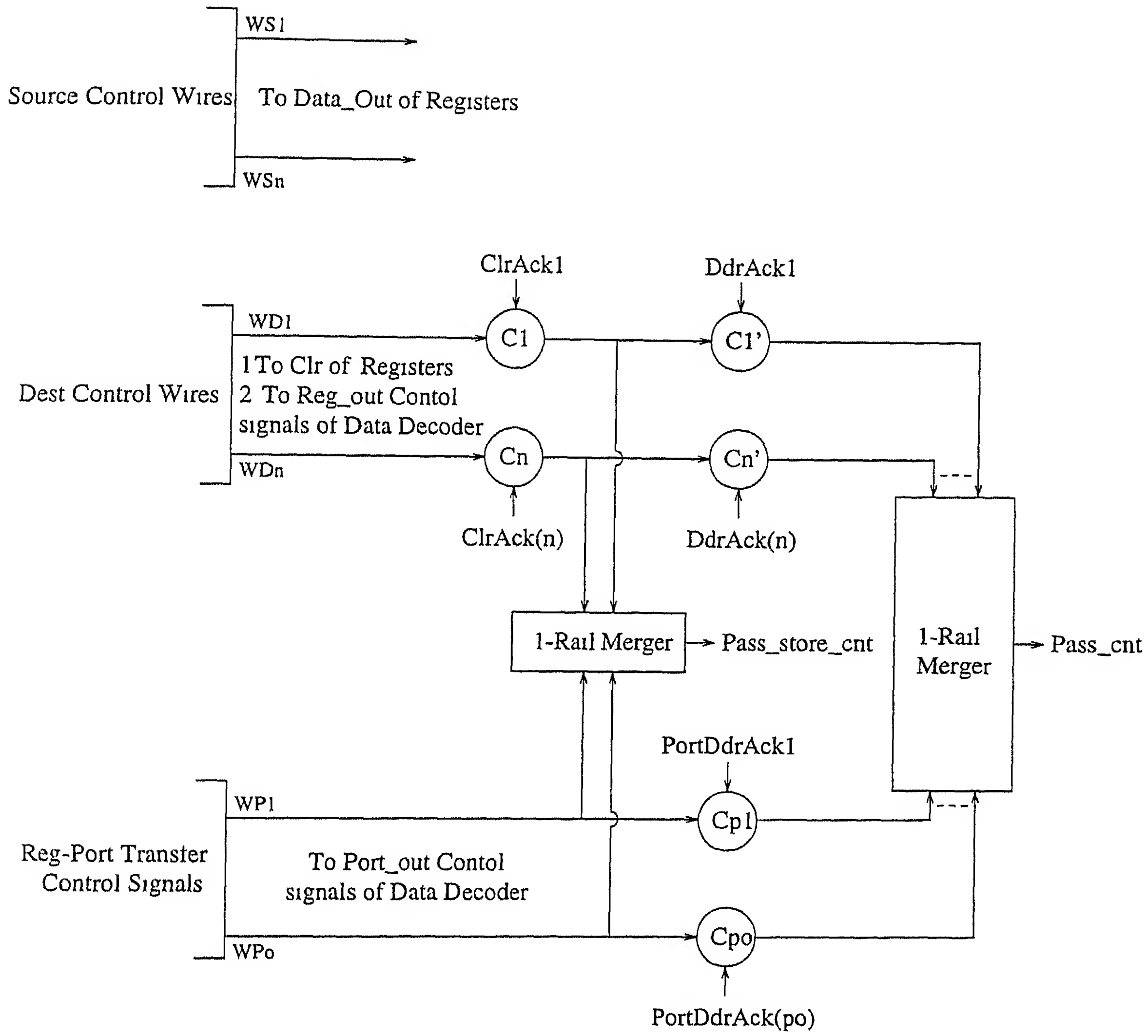


Figure 4 10 Bus Approach 2 Local Bus Controller 2

Module	C Element	XOR Gates
Datapath	$m[6n + 2p_o + 5]$	$m[16n + 2p_i + 5p_o + 9]$
Local bus controller 1	—	$2mn + [(p_o - 1)(m + 1)]$
Local bus controller 2	$m[2n + p_o]$	$2(n + p_o) - 3$

Table 4 3 Bus approach 2 Circuit complexity

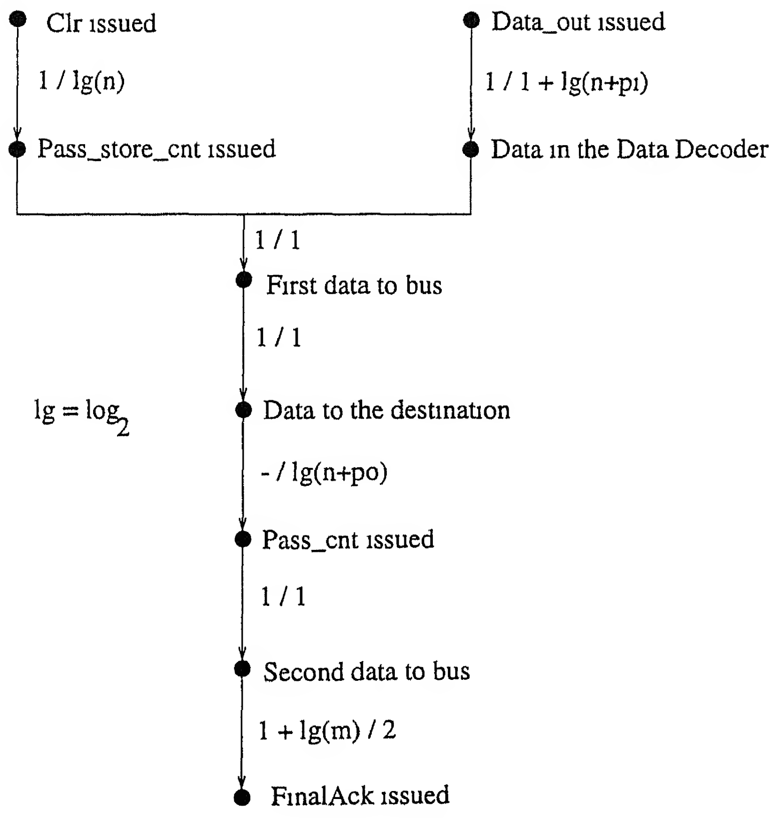


Figure 4 11 Bus Approach 2 Delay in $R_i \longrightarrow R_j$ for local bus controller 1

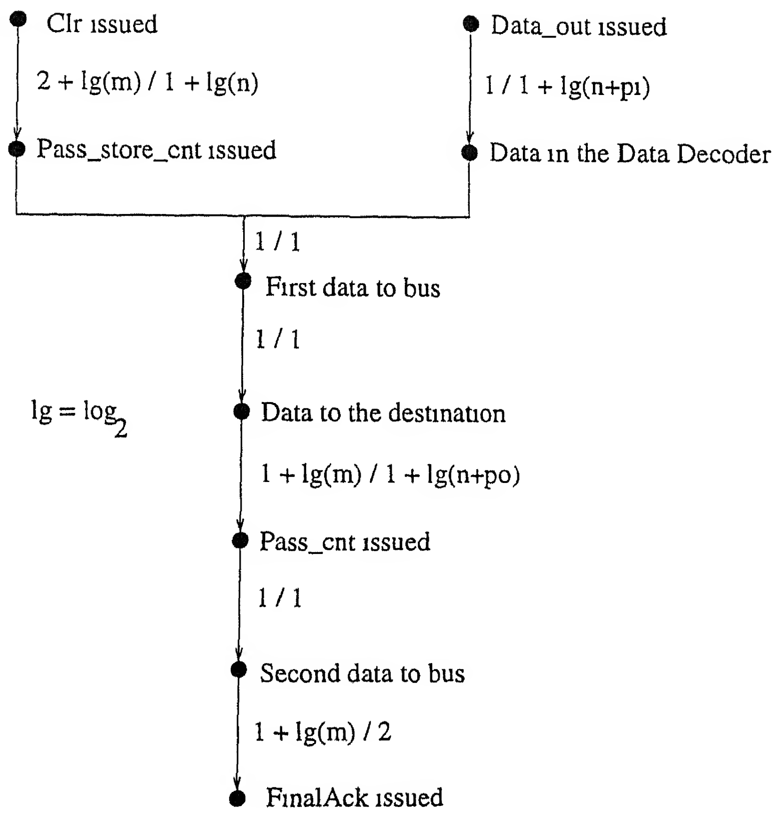


Figure 4 12 Bus Approach 2 Delay in $R_i \longrightarrow R_j$ for local bus controller 2

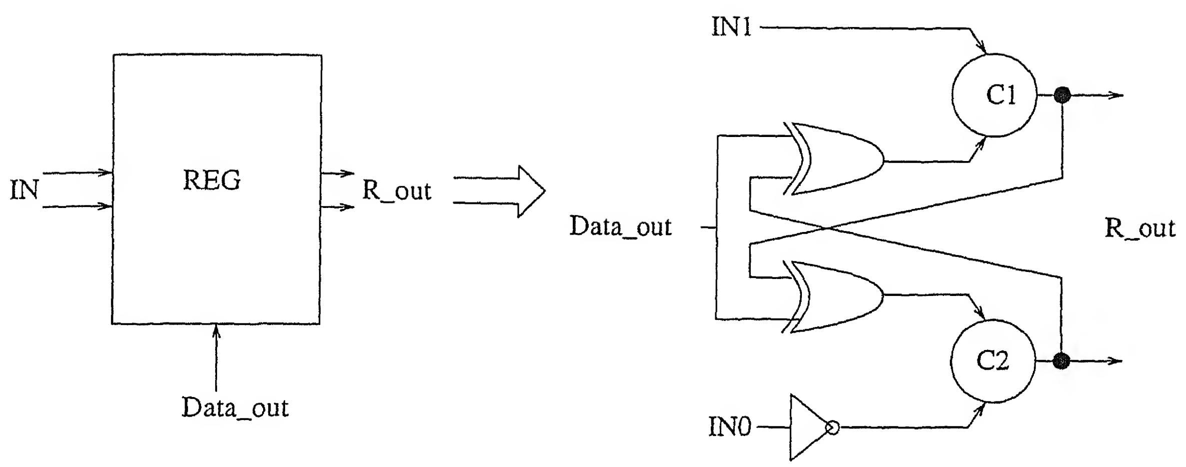


Figure 4 13a Bus Approach 3 Register

4.3.4 Approach 3

The circuit complexity is further reduced by using a different register structure instead of REG1 As shown in Figure 4 13a, this register has a 2-rail input, *In*, a 2-rail output, *R_out*, and a control signal, *Data_out* The stored data is available on *R_out* terminals when *Data_out* is active However, in this process, the stored data is lost On global reset, the register is initialized to a value of 0 because of the inverter As can be seen, the register does not have separate mechanism for clearing the stored data and putting it on the output terminals Therefore, for a register acting as a source, the same data, to be stored, needs to be fed back from the output terminals This is achieved by routing the data bit back to the source register through the data decoder when it is being routed to the destination register To clear the destination register, the stored data is routed to the output terminals which are not connected to the bus

The datapath is as shown in Figure 4 13b The Pass and Store circuit, the Data Decoder and Final Ack generator circuits remain the same as in approach 2 In addition, it uses three 2-rail mergers, A, B and C, a Select block and a U gate The U gate receives the output of merger C It also has two control signals, *Clr_cnt* and *Route_cnt* and 2-rail outputs *Clr_out* and *Route_out* An event on *Clr_cnt* transfers the data at its input to *Clr_out*, which is used to generate a *ClrAck* signal for the controller The data to be cleared is routed to these terminals The *ClrAck* signal

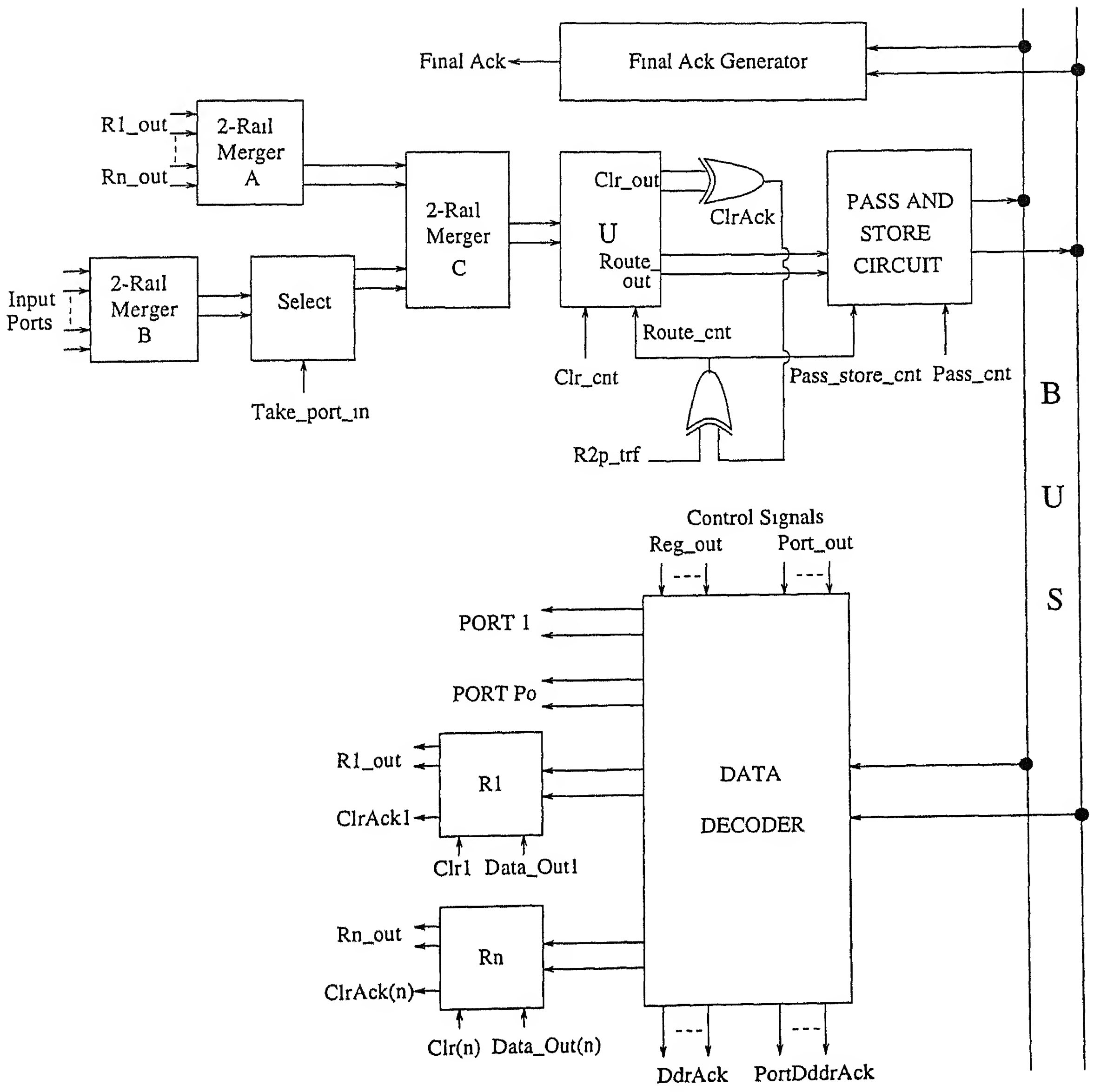


Figure 4 13b Bus Approach 3 Datapath

is used to derive *Route_cnt* for the U gate and *Pass_store_cnt* for the Pass and Store circuit as shown While an event on the *Route_cnt* passes the input to the Pass and Store circuit The 2-rail merger A combines the outputs of all the registers while B combines all the input ports The merger C combines the outputs of A and the Select block

As can be seen, the data bit in R_j which is to be cleared and the useful data bit in R_i which is to be routed to the R_j are both applied to the same input of the U gate Therefore, the following condition needs to be satisfied to avoid possible hazards The data bit, to be routed to R_j, is not applied to U gate until the data in R_j is cleared by the U gate This restriction is satisfied for different modes of data transfer as described below

For $P_i \longrightarrow R_j$, the bus structure does not have control over the time instant at which a new data arrives at the input port Therefore to satisfy the above condition, the new data arriving at the input ports is not applied to the input of the U gate, until the stored data in R_j is not cleared This is achieved by the Select block in Figure 4 13b It receives the data from the 2-rail merger B and passes it to the U gate, only after its *Take_port_in* input is activated by the controller The local bus controller generates this event on *Take_port_in* only after R_j is cleared For $R_i \longrightarrow R_j$, R_j is first cleared The *ClrAck* generated in this process is fed to the controller The controller on receiving it, allows R_i to transfer its data by issuing a *Data_out* signal for R_i This task is realised by the Control Decoder block in the controller For $R_i \longrightarrow P_j$, the restriction need not be satisfied

▮ *Local Bus Controller*

The controller shown in Figure 4 14a, takes the usual inputs as described with respect to the earlier approaches In addition to these, we assume that control signals *R2r_trf* and *R2p_trf* are available Signal *R2r_trf* is active for any $R_i \longrightarrow R_j$, while an active *R2p_trf* signifies a $R_i \longrightarrow P_j$ These signals are assumed to come from the main controller CON1 The Control Decoder consists of the Basic Control Decoder modules shown in Figure 4 14b The principle of operation is same as that of the Basic Data Decoder module excepting for its 1-rail input signal The source

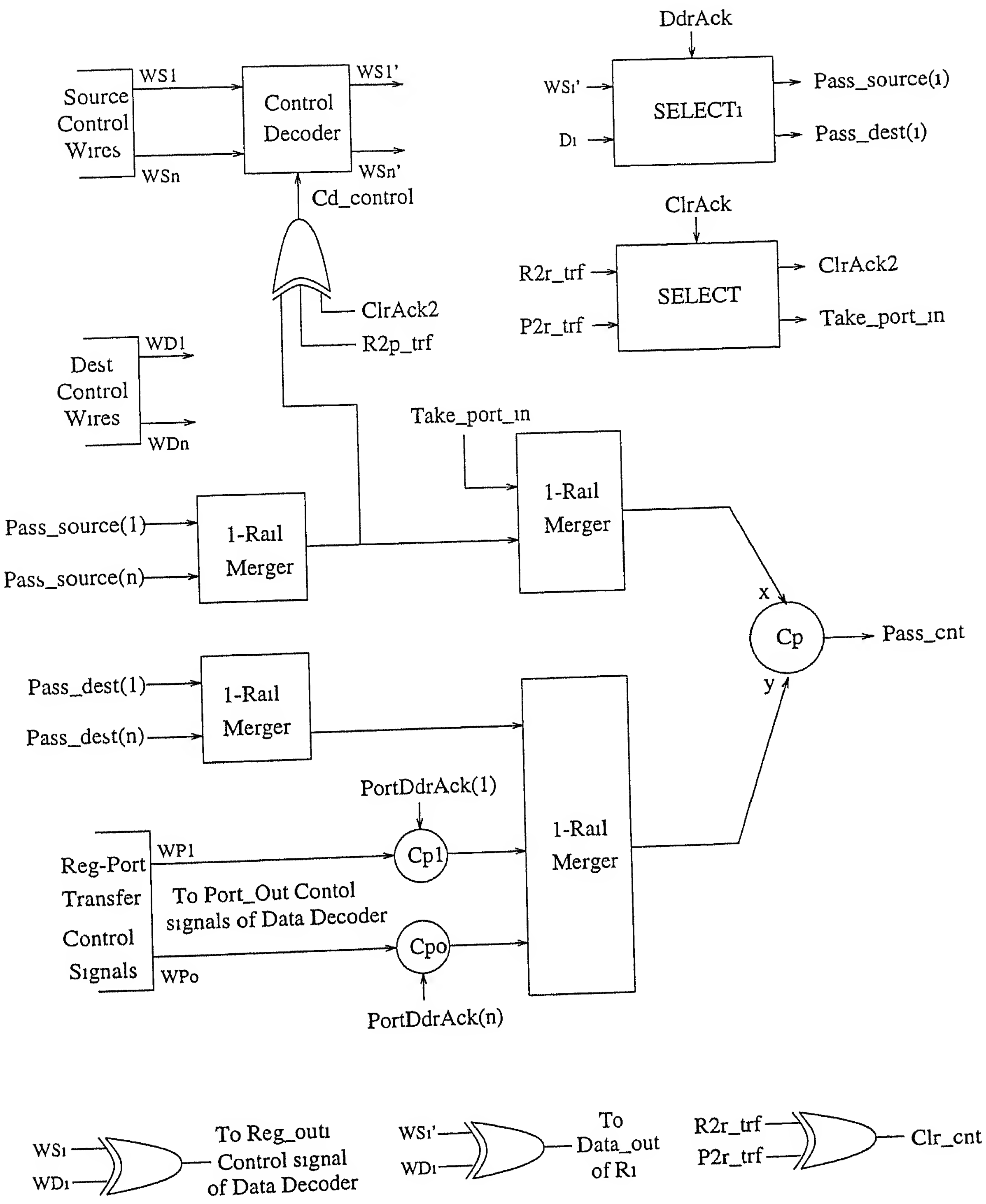


Figure 4 14a Bus Approach 3 Local bus controller

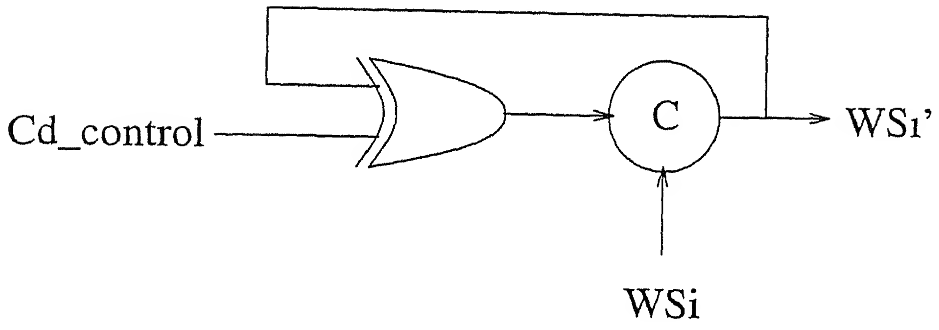


Figure 4 14b Bus Approach 3 Basic Control Decoder Module

control wires WS_1 to WS_n are applied to the Control Decoder It also receives a control signal *Cd_control* It has n outputs $WS_1\prime$ to $WS_n\prime$ For any data transfer having register as a source, two transitions are received by the *Cd_control* wire of the Control Decoder, which are applied to all the Basic Control Decoder Modules The first transition transfers any event that is present in one of the input terminals WS_i to the respective output terminal $WS_i\prime$ The second transition on *Cd_control* cancels the extraneous transitions at the input of the corresponding C element For those modules which do not have any input event, the second transition cancels out the first transition The outputs of the Control Decoder are used to generate the corresponding *Data_out* signals for the source registers

The controller also contains n Select blocks Each Select block, Sel_i, has $DdrAck_i$ from the Data Decoder as one of its inputs $WS_i\prime$ generated within the controller and WD_i are its another inputs It has two single rail outputs $Pass\_source_i$ and $Pass\_dest_i$ A transition on $DdrAck_i$ transfers an event on one of the input terminals $WS_i\prime$ or WD_i to $Pass\_source_i$ or $Pass\_dest_i$ outputs respectively All the n *Pass_source* and n *Pass_dest* signals are used to generate the *Pass_cnt* signal as shown in Figure 4 14a An additional Select block in the controller converts the *ClrAck* generated by the datapath to the signal *Take_port_in* for only the port to register data transfers as shown in the figure For register to register data transfer, the *ClrAck* is converted to $ClrAck_2$

The functioning of the above Select blocks and the circuitry associated with the register to port data transfer will be discussed while describing the actual circuit operation The controller also generates the control signals needed by the datapath elements, e g , *Data_out* for registers, *Clr_cnt* for U gate and Reg_out control signals for the Data Decoder

▪ *Circuit Operation*

$R_i \longrightarrow R_j$ In this, the main controller provides transitions on WS_i and WD_j The event on WD_j clears the contents of R_j An event on *ClrAck* signifies the completion of this The Select block in the controller transfers this event to $ClrAck_2$ This event, in turn, is used to produce a transition on *Cd_control* The Control Decoder then transfers a transition on one of its input WS_i to the corresponding output $WS_i\prime$ This event in turn, produces a transition on *Data_out* of the corresponding source register R_i This causes the stored data of R_i to be transferred to the U gate This data is transferred to the bus through the U gate and Pass and Store circuit as events on both the control signals *Route_cnt* of the U gate and *Pass_store_cnt* of the Pass and Store circuit are already present as a result of a transition on *ClrAck*

The Data Decoder, which already has events on the *Reg_out* control signals of R_i and R_j, enables the data to be transferred to both R_i and R_j Thus R_j gets a new data while R_i retains its original data The Data Decoder produces events on its two *DdrAck* signals corresponding to the data being written to the two registers These control signals are processed by two Select blocks in the controller associated with R_i and R_j, to generate $Pass\_source_i$ and $Pass\_dest_j$ These signals are received by the x and y inputs of the C element, C_p respectively, as shown in the figure, generating a *Pass_cnt* signal Also the $Pass\_source_i$ generates a second event on *Cd_control* input of the Control Decoder, canceling the extraneous transitions in it The *Pass_cnt* signal initiates the remaining set of events, which eventually results in an event on *FinalAck* to signal the completion of the data transfer operation

$R_i \longrightarrow P_j$ For this data transfer, the temporal restriction involving data removal from the destination resource is not necessary The control signals from the main controller drive the wires WS_i and WP_j of the local bus controller WS_i

is applied to the corresponding *Reg_out* control signal of the Data Decoder While WP_j is applied to the corresponding *Port_out* control signal *R2p_trf* feeds the *Route_cnt* input of the U gate and *Pass_store_cnt* of the Pass and Store block It also generates an event on the *Cd_control* of the control decoder, which in turn is responsible for transferring the stored data in R_i to the bus As all the necessary control signals are available, the data is passed to port P_j as well as to R_i

The Data Decoder generates two acknowledges, one on $PortDdrAck_j$ assigned to P_j and the other on $DdrAck_i$ assigned to R_i The latter is processed in the usual way to deposit a transition on the x input of C_p in the Figure 4 14a It also creates the additional transition on *Cd_control* input of the Control Decoder The $PortDdrAck_j$, on the other hand, is received by one input of C element CP_i, the other input of which already has a transition arising out of the WP_j This creates an event on the output of CP_i This event in turn, deposits an event on the y input of the C element C_p Therefore a *Pass_cnt* is generated The next set of events are as described before

$P_i \longrightarrow R_j$ In this, the event on WD_j generates an event on the *Data_out* of R_j The same event is deposited as an appropriate *Reg_out* control signal event in the Data Decoder. An event on port to register transfer control signal *P2r_trf* is used to create the *Clr_cnt* input for the U gate in the datapath This results in clearing R_j The ClrAck generated in this process, in turn generates an event on *Take_port_in* signal in the Select block of the local bus controller An event on *Take_port_in* deposits a transition on x input of C element C_p The same event allows the data at the input port to be transferred to R_j The Data Decoder produces only $DdrAck_j$ here The $DdrAck_j$ is processed by the corresponding Select block and generates an event on $Pass\_dest_j$ wire This, in turn, deposits an event on y input of C_p, generating the *Pass_cnt* signal The rest is same as before

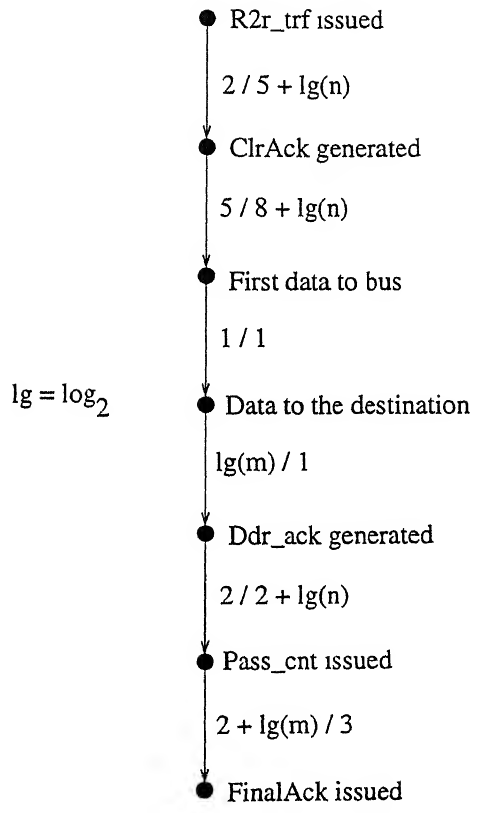


Figure 4.15 Bus Approach 3 : Delay in $P_i \longrightarrow R_j$

Module	C Element	XOR Gates
Datapath	$m[4n+2p_o+11]$	$m[9n+2p_i+5p_o+21]$
Local bus controller	$(n+1)(m+2)+mp_o$	$7n+p_o-3]$

Table 4.4 Bus approach 3 : Circuit complexity

▪ *Complexity*

The circuit complexity in terms of the C elements and XOR gates is as given in the Table 4.4. The delays involved in $P_i \longrightarrow R_j$ are as shown in Figure 4.15.

4.3.5 Delay Constraints for Approach 2 and 3

By the time the *FinalAck* is generated, all the extraneous transitions in the Data Decoder must be canceled. Let the Data Decoder and the Final Ack Generator constitute a block. Then the following delay constraint needs to be satisfied. The delay in the XOR gates used to generate *FinalAck* should be greater than the delays in those XOR gates of the Data Decoder which receive data from the bus. We also assume the existence of an equipotential region in the block.

However, as the number of registers and ports increase, the Data Decoder module becomes bigger. Satisfying the above constraint can then become difficult. In such situations, another version of the Basic Data Decoder module can be used. Using this module eliminates the delay constraint altogether. The Basic Data Decoder module has 2-rail input, 2-rail output and 2 acknowledge signals, *ACK1* and *ACK2*. Its input is connected to the bus and output to the input of a register or a output port. It is shown in Figure 4.16. On the application of *Control*, the first data bit applied is passed to the dotted terminal of the Toggle blocks. This also generates *ACK1*, which can be used as *DdrAck* or *PortDdrAck*. The same is also fed back to the y inputs of C elements C_1 and C_2. The next application of data bit again causes C_1 or C_2 to create a transition on its output. This, in turn produces the data bit on the output terminals. *ACK2* is generated to signify completion of this operation. This signal can be used to generate the *FinalAck*.

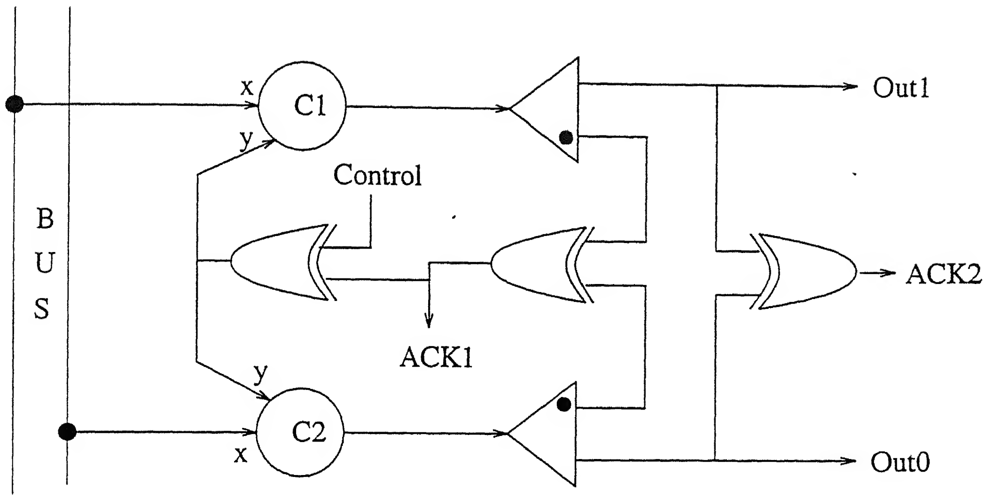


Figure 4.16 DI Basic Data Decoder Module

Module	C Elements	XOR gates
Bus(Approach 1)	3031	11816
Bus(Approach 2-Cont 1)	2664	8078
Bus(Approach 2-Cont 2)	3576	7286
Bus(Approach 3)	2760	4814

Table 4.5 Circuit Complexity Comparison

This Basic Data Decoder module can be easily used in approach 2 and 3 with appropriate changes in the local bus controller. Use of this module leads to an increase in the circuit complexity. However, the increase in the delays will be marginal.

4.3.6 Comparison of the three approaches

For the three approaches, the circuit complexity and delays in $R_i \longrightarrow R_j$ are given in Table 4.5 and Table 4.6 respectively. They are calculated for 50 registers, 8 input and 16 output ports. The data is 8 bits wide.

It is seen that, an attempt to reduce area has resulted in increased delays. It can be noted that the major portion of the delay is contributed by the delays in the 1 or 2-rail mergers, which has logarithmic complexity. Few data transfers, which occur more frequently, can be given higher priority by applying the signals corresponding

Module	C Elements	XOR gates
Bus(Approach 1)	9	8
Bus(Approach 2-Cont 1)	8	19
Bus(Approach 2-Cont 2)	13	12
Bus(Approach 3)	16	38

Table 4.6 Delays in $R_i \longrightarrow R_j$

to them at a latter stage in the merger. This will make these data transfers faster, at the expense of making other infrequent data transfers slower.

It is observed that, for approach 2, controller 1 gives better performance both in terms of number of transistors needed and speed.

4.3.7 $P_i \longrightarrow P_j$ Data transfer

As described before, for data transfers between datapath elements, the bus sees the inputs and outputs of the datapath elements as output and input ports, respectively. Therefore, the data transfer from one datapath element to the other is implemented using $P_i \longrightarrow P_j$.

In approach 1 and 2 for realising the bus structure, this can be achieved by creating an event on the WP_j wire. In approach 3, in addition to this, an additional signal *P2p_trf* is needed, which is active when $P_i \longrightarrow P_j$ is intended. This signal should be disjunctively combined with the other signals, which generate an event on the x input of C_p in Figure 4.14a.

Chapter 5

Issues in Synthesis

In this chapter, issues related to synthesizing asynchronous circuits from HDL descriptions are considered. We assume that asynchronous circuits will be designed using our approach. The issues related to synthesis in the synchronous domain are well known. We refer [14, 15] for the synthesis of synchronous digital circuits.

In trying to adapt the hand synthesized design style to the task of automated synthesis from HDL descriptions, we observed that, any asynchronous circuit implementing a design can be represented using two distinct blocks; controller and datapath which interact as shown in the Figure 5.1. Furthermore. a natural hierarchy of controllers was seen to be present in most designs. The controller is split into the main controller and the local controller embedded within the datapath. The controller is split because of the following reasons.

- The control information in the CDFG or CFG can be used directly to derive the main controller.
- The local controller is derived from the DFG after scheduling and binding operations have been carried out on it. It also uses the information related to scheduling and binding to derive the specifications for the interconnection network.

The main controller issues control signals $C_1, \ldots, C_n$ to the local controller and receives a corresponding acknowledge for each of them. The datapath generally consists of the resources like datapath elements and registers interconnected through the

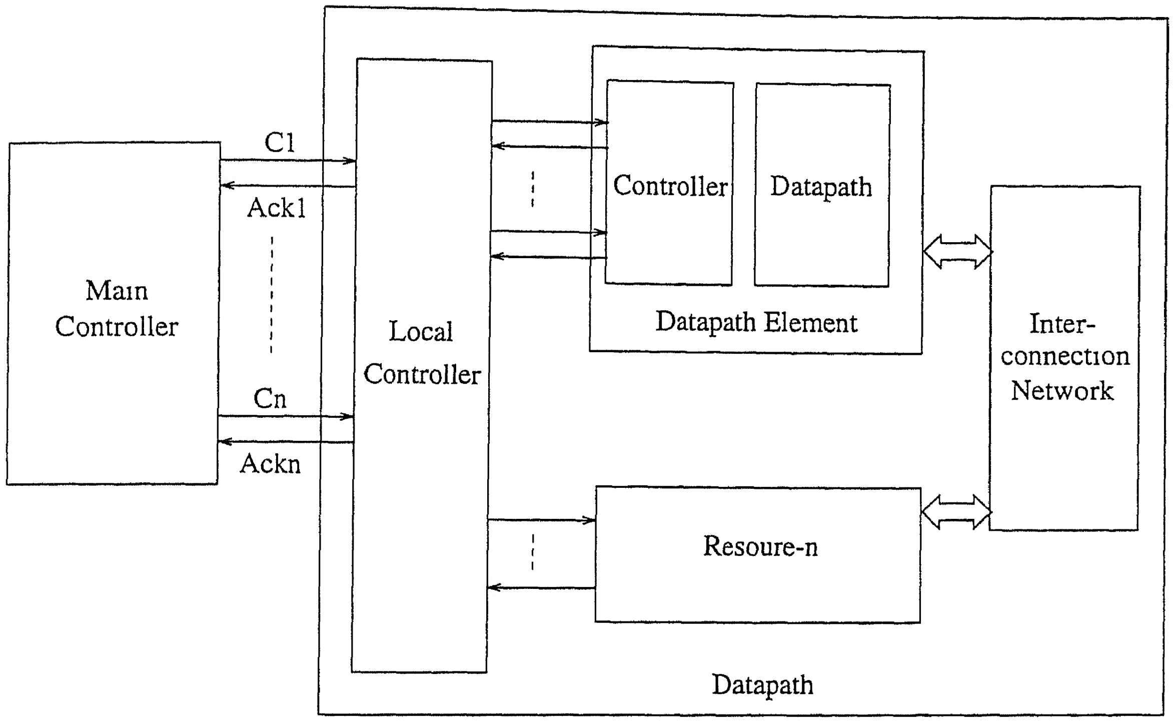


Figure 5 1 Block diagram of the synthesized design

interconnection network Data transfer between the resources is realised under the direction of the local controller using the elements in the interconnection network The datapath element, in turn, can have their own local controller to internally coordinate various events necessary for carrying out their intended functionality

5.1 Synthesis Outline

The HDL description is processed to get a Control Data Flow graph (CDFG) The CDFG is further divided into a Control Flow Graph (CFG) and a Data Flow Graph (DFG) The CFG and DFG are then used to carry out the remaining synthesis process The synthesis task essentially consists of four distinct phases

1 Scheduling

2 Resource sharing and binding

3 Interconnections and storage allocation

4 Control unit generation

Out of these, the first two are similar to that in the case of synchronous synthesis However, the remaining two steps differ significantly We briefly describe each of the phases below

5.1.1 Scheduling

In the synchronous design approach, scheduling is the partitioning of the design behavior into control steps such that all operations in a control step execute in one clock cycle However since there is no clock in asynchronous designs, an alternate definition of the scheduling task is as below [15]

Let the execution delays of the operations in a DFG be denoted by $d_1, \quad , d_n$ We define the start time of an operation as the time at which the operation starts its execution Let the start times of the operations are represented by $t_1, \quad , t_n$ Scheduling is the task of determining the start times, subject to precedence constraints specified by the DFG

Besides the functional behavior of any datapath element, we associate with it additional parameters viz , its area on silicon and the average and worst case delays In the design paradigm chosen, any datapath element can take an arbitrary time to complete its operation correctly However, scheduling the DFG with respect to the above mentioned delays can result in an implementation with maximum concurrency and sharability of resources, leading to the improved performance

In the synchronous paradigm, the operations are scheduled in a number of discrete and equal time steps, the time step being decided by the worst case delay amongst the various functional units for unit step functional elements or the best case delay for multistep functional units In asynchronous domain, there in no notion of a clock However, the various operations can still be scheduled over time steps The maximum time step for any DFG is the minimum of the individual delays of all the datapath elements Smaller the time step, better is the schedule obtained However, there is no point in reducing the time step below the minimum of the delays in the basic elements used e g C element, XOR gate and inverter Also, One can use different time steps for different DFG's in the same system

With the above assumption, we can use the existing scheduling algorithms in the synchronous domain Scheduling algorithms in the synchronous domain have been divided into two broad classes depending upon the constraint used to drive the scheduling task

Resource constrained scheduling In this, there is an upper limit on the number of datapath elements that can be used The scheduling is done by assigning the average case delays to the datapath elements instead of the worst case delays This is possible because, in the design methodology adopted, the circuit implementation works correctly, irrespective of the delays in the various circuit blocks Therefore, worst case delays need not be assigned to ensure the same Using the average case delays increases the possibility of better performance

Time constrained scheduling In this, there is an upper limit on the execution time of the CDFG Therefore, the worst case delays must be assigned to the datapath elements Furthermore, the delays in the interconnection network and in the controller should also be accounted for Also, the assumption that, any circuit block can have an arbitrary execution time, will no longer be valid

5.1.2 Resource Sharing and Binding

This is nothing but assignment of operations, memory accesses, and interconnections from the behavioral description to hardware units for optimal area and performance

In our case, this task is mainly dictated by the fact that the output of each datapath element and data stored in each register is transferred to the other resources through the corresponding interconnection elements in the interconnection network The area complexity and the delay associated with these interconnection elements increase with the number of destinations, P_0, that each source resource has

Binding Objective : If resources are shared such that, for all n interconnection elements, $\sum_{i=1}^{n} P_{o_i}$ is minimized, we get a minimized interconnection network

5.1.3 Interconnections and storage allocation

The resources can be interconnected using either the point to point topology or the bus topology In the point to point topology, the DFG obtained after scheduling and binding, is used to generate the interconnection network consisting of the interconnection elements In our case, the interconnection elements can also serve as storage units In the approach based on the bus topology, the interconnection network topology is more generic and is generally fixed apriori Only the registers need to be allocated to store constants and variables In both the cases, a set of rules is applied to ensure a hazardless implementation of the data transfers implied by the scheduling and binding done with respect to a given DFG

5.1.4 Control unit generation

As stated before, the main controller is derived directly by using the CFG while the local controller is obtained by using the information derived in the above step and the DFG It is evident that the main controller derived in synthesis for both the interconnection topologies will be identical The local controller for both the topologies carries out the data transfers between the various resources Data transfers in the point to point topology can be concurrent, but those on a single bus structure will be necessarily sequential This fact and the implementation differences in both the topologies make the realisation of the local controller different for the two topologies

5.2 Synthesis for Point to Point Interconnection Topology

Consider a differential equation integrator, the behavioral description of which is given below

while $(x < a)$ *do*

$$x_1 = x + dx,$$

```
        $u_1 = u + 3xudx - 3ydx,$
        $y_1 = y + udx,$
        $x = x_1,$
        $y = y_1,$
        $u = u_1,$
    endwhile
```

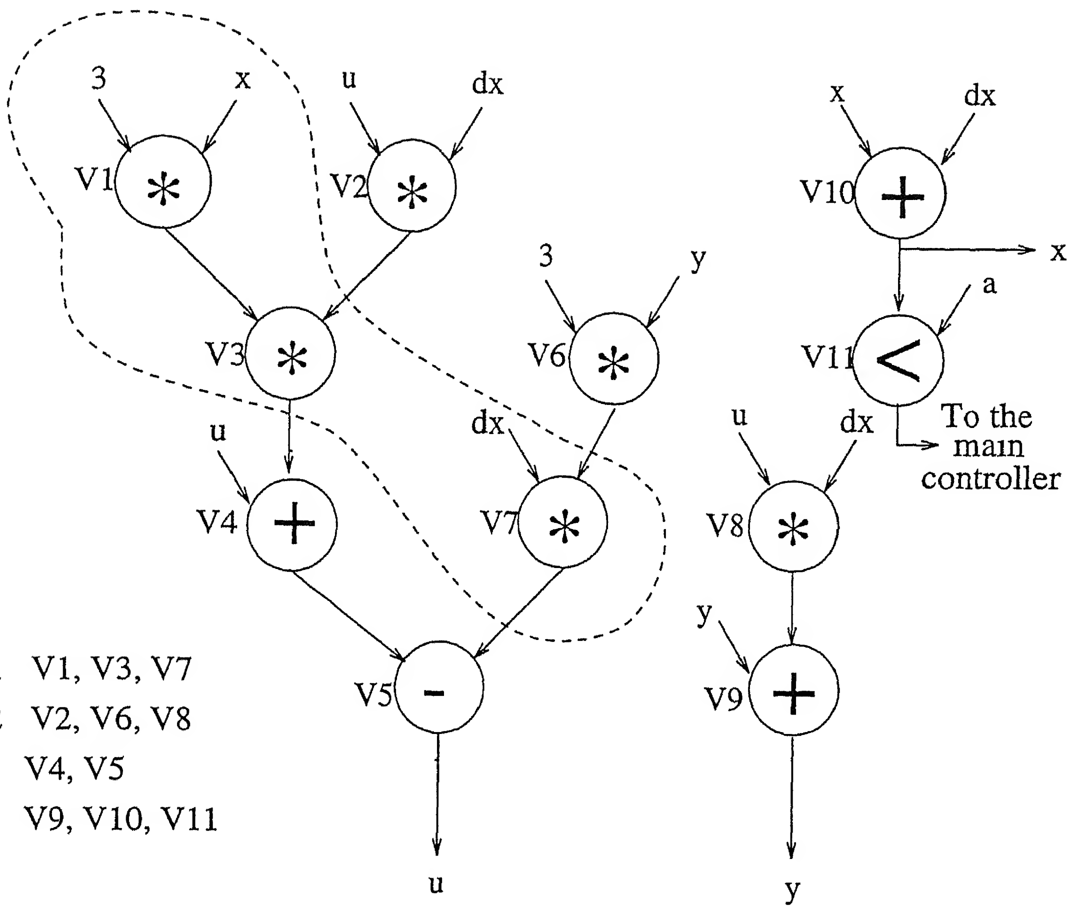
Figure 5 2 shows the DFG after scheduling and binding have been performed It has 11 operational nodes The implementation employs two multipliers and two ALU's The operations indicated by the DFG are executed iteratively For the sake of simplicity, it is assumed that the delay in ALUs and Multipliers are identical and is equal to the time step used to carry out the scheduling task

As can be seen, that the data that each of the four datapath elements receive at their inputs can be grouped into three classes

- Output of any of the datapath elements (e g , output of ALU1, etc)
- Constants (e g , 'a', 'dx' and '3')
- Variables that are stored across iterations (e g , 'x', 'y' and 'u')

The data belonging to all the three classes can be stored in general purpose registers implemented using UReg or REG1, which can then be transferred to the appropriate destinations However, it is seen that, by using three different interconnection elements for these classes of data, circuit complexity and performance can be improved The output of the each datapath element is directly transferred to the required destinations through their corresponding A_Demux element This has been discussed in detail in the last chapter As the A_Demux also stores the output of the datapath element, till it is transferred to the destination, we call it 'A_Demux_Store' For constants, we use 'Const_Demux_Store' While for the third data type, an element called 'Iter_Var_Store' is used The implementation of these elements is given later

Let the DFG consists of m vertices representing m operations We assume that these operations are carried out by n datapath elements, $n <= m$ Each datapath



MUL1	V1, V3, V7
MUL2	V2, V6, V8
ALU1	V4, V5
ALU2	V9, V10, V11

Figure 5 2 DFG of Differential Equation Integrator

element generates an acknowledge to indicate the completion of its operation Also, Iter_Var_Store generates an input acknowledge each time a new data is written into it by its source datapath elements The control signals necessary to coordinate the execution of the m operations and the data transfers in a DFG are generated by the local controller using these acknowledge signals and the control signals from the main controller To carry out the above task, the local controller may also need a few input acknowledge signals from some of the datapath elements The input and output acknowledge signals for a 2-input, single output datapath element, generated while executing node V_i in a DFG, are indicated by an A_iI_1, A_iI_2 and A_iO respectively While the *ith* input acknowledge signal for an Iter_Var_Store element R is indicated by AR_i

5.2.1 Data Transfer Guidelines

We use the following notation for the discussions below DPE_i represents a particular instance of a datapath element or a functional unit By RES_k we mean the kth resource, where a resource signifies one of the following, a datapath element, a storage unit or an interconnection unit

Assume that a datapath element DPE_i has performed its current operation V_x and its output is transferred to $A\_Demux\_Store\_i$ This data is to be transferred to one of the inputs of the datapath element DPE_j, to start a new operation V_y of DPE_j Assume that the last data written into the $A\_Demux\_Store\_j$ needs to be transferred to another resource RES_k Then the following is true

1 If each output of DPE_i is always transferred to the same input of DPE_j and this input of DPE_j receives the output of DPE_i only, then the corresponding $A\_Demux\_Store_i$ can be replaced with wires

2 Data from A_Demux_i is not transferred to DPE_j unless the following conditions are satisfied

   (a) The DPE_j has completed its previous operation This condition needs to be satisfied to avoid hazards at the corresponding input of DPE_j This is also true for $i = j$

   (b) The previous output of DPE_j has been transferred to RES_k If data is transferred to DPE_j from $A\_Demux\_Store_i$ without satisfying the above condition, the next output of DPE_j may create hazards at the input of $A\_Demux\_Store_j$

   An event on the output acknowledge A_jO of DPE_j, prior to the desired data transfer, signifies that the first condition can be met satisfactorily, while that for the second condition is signified by an event on the input acknowledge, AK_i, of RES_k

Special Cases

(a) If the data in $A\_Demux\_store_i$ is to be transferred to DPE_i, then both the conditions need not be satisfied In this case, there in no possibility of hazard at all However, if A_Demux_i has multiple outputs, the output acknowledge A_xO is used to transfer the data back to DPE_i

(b) If previous output of DPE_j is used by itself in V_y, only the first condition needs to be satisfied In this case, the DPE_j will receive data from $A\_Demux\_store_j$ at its one input and from $A\_Demux\_store_i$ at its other input

(c) If the data is being written into the DPE_j for the first time in the current execution of DFG, both the conditions need not be satisfied However, as in the first special case , output acknowledge A_xO of DPE_i is used to initiate the data transfer

3 Deadlock condition If RES_k is DPE_i, then a deadlock condition can occur This is illustrated with an example Figure 5 3 shows a DFG with four nodes An adder and a multiplier are employed to execute these four operations Figure also shows the A_Demux_Store blocks for the adder and the multiplier obtained by applying the above stated rules It can be seen that, the signals A_4I_1 and A_3I_1 are never generated and a deadlock condition occurs In order to avoid this, one of the above mentioned signals is not used

4 If current output of DPE_i is to be used later by any datapath element, after one or more operations are done by DPE_i, it needs to be stored Figure 5 4 shows such a situation in a DFG employing an adder and a multiplier The modified A_Demux_Store for the multiplier is as shown in the figure

5 If the output of DPE_i is to be transferred to a Iter_Var_Store, the corresponding signal to the $A\_Demux\_Store_i$ is not issued unless the earlier data stored in the corresponding Iter_Var_Store is cleared An implementation of this is specific to a particular realisation employed for the Iter_Var_Store element ,

6 The control signals to the $A\_Demux\_Store_i$ obtained by the application of above rules may occur concurrently if there is no temporal relation between

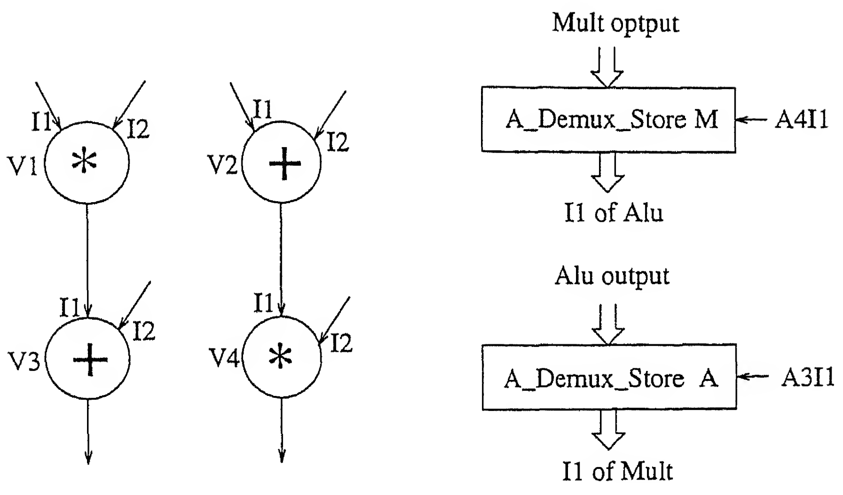


Figure 5 3 Deadlock condition

them This will create malfunction in *A_Demux_Store*$_i$ This can be easily avoided as described next Let the control signals C_p and C_q occur concurrently C_p and C_q transfer the output of DPE_i generated while executing V_p and V_q respectively Then C_p should be combined with A_pO using a C element, before applying it to the *A_Demux_Store*$_i$ In a similar way, C_q should be combined with A_qO The control signals to the other interconnection elements Const_Demux_Store and Iter_Var_Store can be applied concurrently

5.2.2 The Local Controller

For a DFG with m vertices, the local controller decodes the output acknowledges of n datapath elements into m signals A_1O to A_mO This is done using n Event Counters as described next Let a datapath element DPE_i execute the operations in vertices V_p, V_q and V_r of DFG The output acknowledge signal of DPE_i is applied at the *EventIn* input of a modulo-3 Event Counter to derive A_pO, A_qO and A_rO at its outputs. If a few of the input acknowledges A_iI_1 and/or A_iI_2 , $i = 1, \quad , m$, are needed, then by adding the necessary circuitry at these inputs of datapath elements, we generate the input acknowledges from the datapath elements These signals are then decoded using Event Counters to get the necessary signals

Controller uses the above signals along with the control signals from the main

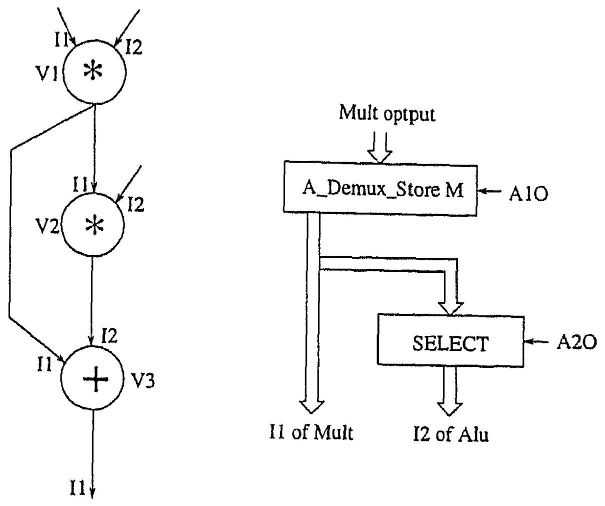


Figure 5 4 Modified A_Demux_Store

controller to generate control signals for the datapath elements and interconnection network It also generates acknowledge signals for the main controller using the same signals

5.2.3 Interconnection Elements

Depending on three different classes of data the datapath element receives, three interconnection elements are needed to store and transfer the data viz , A_Demux_Store, Const_Demux_Store and Iter_Var_Store Out of these, A_Demux_Store is nothing but A_Demux as described at the start of this section The remaining two elements are discussed here

These elements are described with the help of DFG shown in Figure 5 5a V_i, V_j, and V_k are the three operational nodes in the DFG, which receive the data represented by DAT Further, the following is assumed These nodes are executed by DPE_i, DPE_j and DPE_k respectively The data DAT is transferred to these datapath elements using the control signals C_i, C_j, and C_k respectively The DFG is executed in iterations and each iteration is initiated by the controller through an event on *St_iteration* control signal *Done* is the control signal indicating the end of the loop

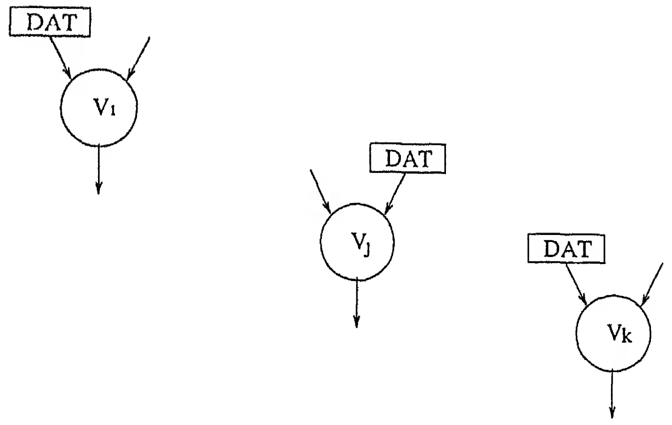


Figure 5 5a Vertices in a DFG receiving the same data DAT

■ *Const_Demux_Store*

Let DAT be a constant used in the execution of a loop Also, let all the operations V_i, V_j, and V_k be executed in each iteration Then Const_Demux_Store element for DAT is as shown in Figure 5 5b On each application of *Start_iteration*, the data in UReg is made available at its *Data_store* terminals This data can then be transferred to the desired datapath elements using either C_i, C_j, or C_k The application of control signal *Done* clears the data in UReg by transferring it on the *Out* terminals The same task could have been achieved by replacing the Select blocks with an A_Demux block However, this results in greater circuit complexity Also, in this implementation, concurrent application of C_i, C_j and C_k is possible This may improve performance

If the execution of V_j and V_k is conditional in individual iterations, then the Const_Demux_Store is realised as shown in the Figure 5 5c Here, C_{j_k} is the control signal generated by the controller when C_j or C_k are to be executed

■ *Iter_Var_Store*

Let DAT be a variable which is used across iterations and is also updated in each iteration Then the Iter_Var_Store corresponding to DAT is shown in Figure 5 5d An event on *Start_iteration* transfers the data in the U-gate to the Select blocks, which can then be routed to the datapath elements After this, no data exists in the

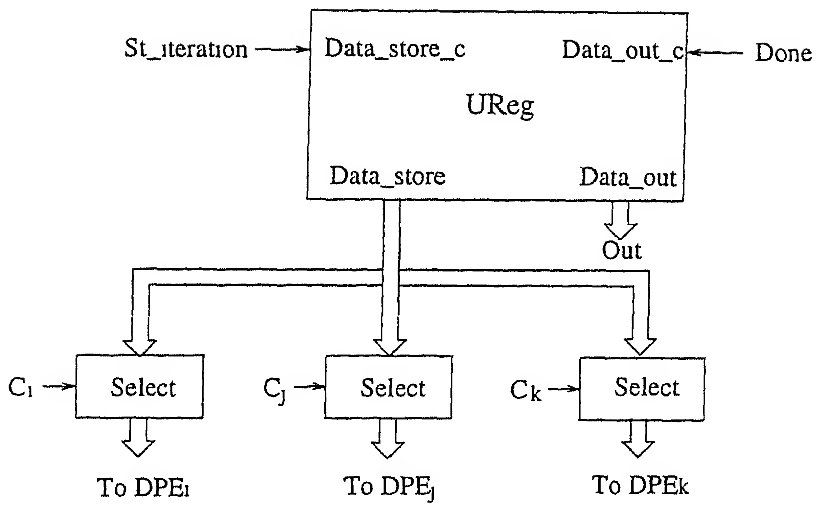


Figure 5 5b Const_Demux_Store

U-gate As a result of execution of DFG, a new data value is written in the U-gate, which is used in the next iteration Activation of the *Done* signal transfers the data updated and stored in the U-gate in the last iteration, to the *Out* terminals

Consider that the execution of V_j and V_k is conditional The Iter_Var_Store can have different realisations depending on how the variable DAT is used Two typical cases are discussed below

- Let DAT be updated by the executions of V_i and either V_j or V_k Figure 5 5e shows the Iter_Var_Store for this case

- Let DAT be updated by the execution of V_i only Then the corresponding implementation is as shown in Figure 5 5f When the data is to be transferred to DPE_j and DPE_k, it should also be retained, so that it can be transferred to DPE_i when needed in the next iteration Therefore, an UReg is employed The particular version of Iter_Var_Store needed to achieve this for different cases can be similarly derived for any DFG

5.2.4 Examples

We employ the above ideas to derive asynchronous implementations from the HDL descriptions of the Shift Multiplier and the Differential Equation Integrator

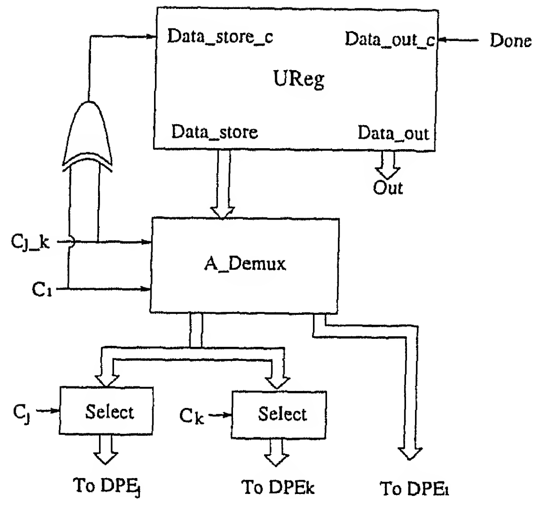


Figure 5 5c Const_Demux_Store to support conditional execution of V_j and V_k

■ *Shift Multiplier*

In the Shift Multiplier the multiplicand is represented by variable B and the multiplier by A The final result is indicated by M

The scheduled CDFG for the Shift Multiplier [14] is as shown in Figure 5 6

The main controller is obtained directly using the control flow information in the CDFG and is shown in Figure 5 7a The loop consisting of four iterations is implemented using the mod-5 Event Counter The main controller generates control signals *Add* and *Shift* for the local controller to carry out the implied functionality It also issues the signal *Done*, which signifies the end of the multiplication operation for a given set of input instances It receives *AddAck* and *ShiftAck* from the local controller

The DFG implementing the multiplication operation has three vertices V_1 to V_3, as shown in Figure 5 7b Let an adder and a two shifters SHR1 and SHR2 be assigned to these operations The interconnection elements $A\_Demux\_Store_1$ to $A\_Demux\_Store_3$ for these datapath elements are derived and are shown in Fig 5 7c Consider $A\_Demux\_Store_1$ and $A\_Demux\_Store_2$ The outputs of both are transferred to the same destination, which stores M. Also, the two control signals A_1O and A_2O can never occur concurrently Therefore, a single A_Demux_Store can be used instead of two, as shown in Fig 5 7e

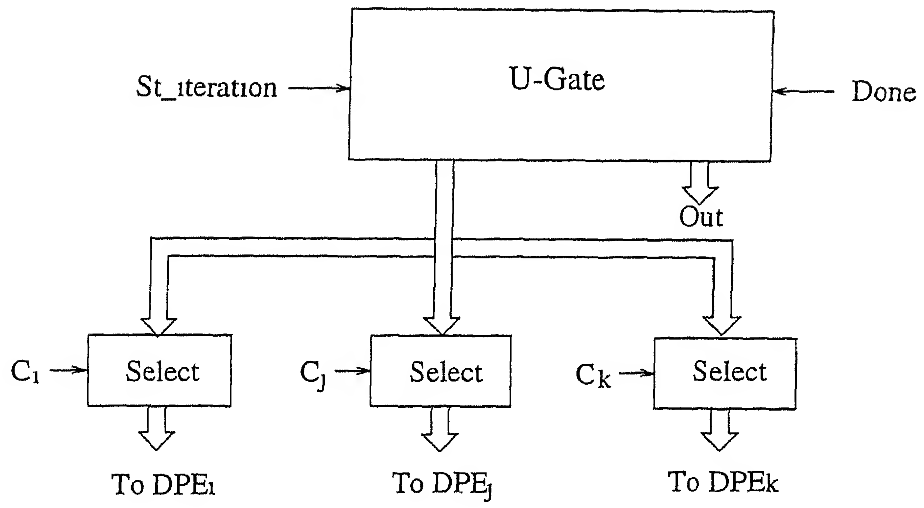


Figure 5 5d Iter_Var_Store

It can be seen from the DFG that B is a constant and it is applied to the adder once in each iteration So the corresponding Const_Demux_Store takes the form of a UReg Variables A and M are needed across iterations The corresponding implementation of Iter_Var_Store for the variable M is as shown in Figure 5 7d, while that for variable A takes the form of a U-gate

All the circuit blocks are interconnected to get the datapath shown in Figure 5 7e It can be seen that, the Start signal is used to initialize A, B and M Also the *Done* signal generated by the main controller is used for transferring the product available in Iter_Var_Store modules corresponding to variables A and M The local controller is derived as described before and is shown in Figure 5 7f

▮ *Differential Equation Integrator*

The behavioral code and the DFG for the Differential Equation Integrator after scheduling and binding is given earlier in Figure 5 2 The ALU used in this example performs addition, subtraction and comparison It has two 2-rail inputs and three 1-rail control signals *Add*, *Sub* and *Comp* The result of addition and subtraction is made available on its 2-rail output terminals While the result of comparison operation is made available on one of its three 1-rail outputs, namely *a_ge_b*, *a_le_b* and *a_eq_b* The completion of each operation is signified on its output *AluOutAck*

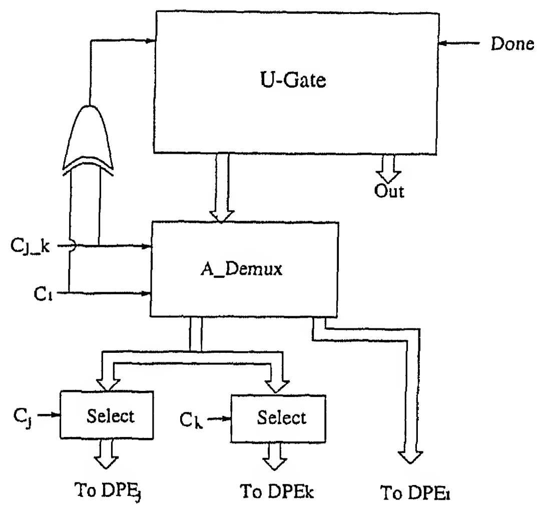


Figure 5 5e Iter_Var_Store Case 1

In addition to input, output and output-acknowledge terminals, the multipliers also have a control signal *Multiply*, which initiate the multiplication operation

The synthesis process, as outlined below, starts with the derivation of the main controller The main controller is as shown in Figure 5 8a Execution of each new iteration is initiated by creating an event on *St_iteration* control signal The completion of execution of each iteration is signified by an event on the *ItCompAck* wire The main controller also receives a 2-rail signal *X_ge_A* from the datapath If the value of this signal is 1, controller initiates the next iteration

The interconnection network is derived next First, the A_Demux_Store for the multipliers and the ALUs is obtained by applying the set of rules described earlier They are shown in Figure 5 8b along with the datapath elements that they are connected to The following can be observed

- The output of multiplier MUL_1 generated while executing either V_3 or V_7 is transferred to the same input of ALU_1 This results in a less complex A_Demux_Store element which has two outputs instead of three The same is true for multiplier MUL_2

- Consider A_Demux_Store element corresponding to MUL_2 The control signals $A_{11}O$ and A_4I_2 may occur concurrently Therefore, A_6O is combined with A_4I_2, and A_8O with $A_{11}O$, using C elements, as described in Section 5 2 1

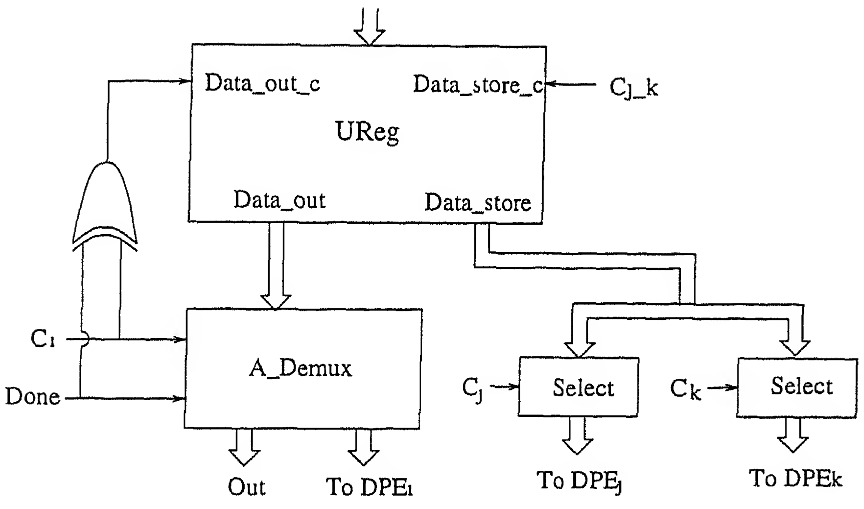


Figure 5 5f Iter_Var_Store Case 2

The interconnection elements Const_Demux_Store for the constants a, dx and 3 are shown in Figure 5 8c The realisation of all of them is very similar to that shown earlier in Figure 5 5b

Figure 5 8c shows the Iter_Var_Store elements for all the three variables x, y and u Their realisation is similar to that shown earlier in Figure 5 5d

Finally, we describe the local controller implementation It is shown in Figure 5 8d It employs Event Counters to generate almost all the control signals needed for the interconnection network It also generates the control signals *Add*, *Sub*, *Comp* and *Multiply* for the ALUs and multipliers respectively The signal *ItCompAck* is generated using the input acknowledge signals of the three Iter_Var_Store elements corresponding to the variables x, y and u The signal *X_ge_A* which is sent to the main controller is derived using the 1-rail outputs of ALU2

5.3 Synthesis for Bus Interconnection Topology

In a digital system, let the number of resources amongst which, data transfer needs to be carried out increase In such cases, the point to point interconnection topology may result in increased cost of realisation in terms of area, and can also result in degraded performance due to increased delays To reduce this area cost, we can instead realise these data transfers using the bus topology

In the discussion that follows, the design process and the synthesis issues based on the bus topology are highlighted The basic idea is illustrated through three examples wherein we also give methods of improving the performance in the bus topology The first design is that of a Shift Multiplier The second design is based on the same multiplier implemented using a combination of point to point topology and the bus topology Here we highlight the use of a multi-bus structure The final design is that of a Differential Equation Integrator Through this example, we illustrate an efficient method of using the bus structure, by appropriately ordering the data transfers We wish to make it clear that for these examples, the point to point topology results in better implementations in terms of both area and performance However, our focus here is on the design process and the synthesis issues

5.3.1 Example 1 : Shift Multiplier

We consider the same shift multiplier discussed with respect to the implementation based on the point to point interconnection topology We give its implementation using the bus topology As before, the resources needed to implement the shift multiplier are an adder, a shifter and registers to store the variables, A, B and M Figure 5 9a shows the abovementioned resources symbolically connected to the bus The actual circuitry and the local bus controller are not shown for the sake of simplicity

Unlike registers, the datapath elements do not contain any data when they are idle Therefore, prior to transferring data to, we do not need to clear them The data to their inputs is transferred using either a $R_i \longrightarrow P_j$, or a $P_i \longrightarrow P_j$ mode of data transfer Their inputs can, therefore, be treated as output ports Hence, these are named as $P\_ADD\_I_1$, $P\_ADD\_I_2$ etc The output of each datapath element is stored in an associated Select block On the application of an event on its single rail control input, the stored data is transferred to its output These outputs are connected to the bus The data from the Select blocks are transferred to the destination registers using $P_i \longrightarrow R_j$ data transfer mode Therefore, these outputs act as input ports for the bus structure and are labeled accordingly, e g

Main controller signals	Local controller signals	Operation
Start	Start	$A\_Port \longrightarrow A\_Reg$
	Start2	$B\_Port \longrightarrow B\_Reg$
	Start3	$0 \longrightarrow M\_Reg$
Add	Add	$M\_Reg \longrightarrow P\_ADD\_I_1$
	Add2	$B\_Reg \longrightarrow P\_ADD\_I_2$
	Add3	$P\_ADD\_OUT \longrightarrow M\_Reg$
Shift	Shift	$M\_Reg \longrightarrow P\_SH\_I_1$
	Shift2	$A\_Reg \longrightarrow P\_SH\_I_2$
	Shift3	$P\_SH\_OUT_1 \longrightarrow M\_Reg$
	Shift4	$P\_SH\_OUT_2 \longrightarrow A\_Reg$
Done	Done	$M\_Reg \longrightarrow P\_out\_msb$
	Done2	$A\_Reg \longrightarrow P\_out\_lsb$

Table 5 1 Data transfers in Single-bus Shift Multiplier

P_ADD_OUT, $P\_SH\_OUT_1$, etc The completion of operation performed by each datapath element is signified by an event on its Ack output, e g *AdderAck*, *Shr1Ack* etc After completion of multiplication, the product is sent out to the output ports *P_out_lsb* and *P_out_msb*

The main controller derived from CFG is same as that of the shift multiplier designed before, except for the inclusion of two more C elements as shown in the Figure 5 9b These C elements are included to acknowledge the completion of two sets of data transfer, each initiated by the *Start* and the *Done* signals, as shown in the Table 5 1 In this Table, each control signal C_i issued by the main controller, is used by the local controller to invoke a sequence of data transfers to implement the desired functionality implied by C_i Let n data transfers be needed to achieve this Then the local controller generates n control signals sequentially Excepting for the first, each of them is generated only after an appropriate acknowledge is received from the bus

Table 5 1 shows the signals issued by the local controller in response to each control signal sent by the main controller, and the corresponding data transfer to be implemented using each of them Two examples of data transfer implementation follows

- Control signal $Start_2$ should be used as a control signal for the 1-2 Converter associated with port B The same should be used to generate an event on the appropriate destination control wire of the bus structure, which in turn generates the *Clr* input of B_Reg to clear it

- Similarly, *Add* is used to generate an event on the WS_i control signal associated with the M_Reg and WP_j associated with the $P\_SH\_I_1$ The generation of the bus control signals from those of the local controller can be done using XOR gates, e g , WS_i corresponding to the A_Reg is derived by disjunctively combining the $Shift_2$ and the $Done_2$ control signals

Figure 5 9c and Figure 5 9d shows how the local controller generates the control signals listed in column 2 of Table 5 1 It uses signals from the main controller, the acknowledge signals signifying the completion of individual data transfers and the output acknowledge signals of the datapath elements A single *FinalAck* is decoded into 12 different acknowledge signals by the local controller as shown in Figure 5 9c Out of these, 4 signals are passed to the main controller and remaining signals are used by the local controller A Control Decoder and several Event Counters are employed to generate these signals as shown in Figure 5 9d

5.3.2 Example 2 : Shift Multiplier : Mixed Approach

The datapath is as shown in Figure 5 10a In this example, A_Reg, one input of shifter $P\_SH\_I_2$, one output of the shifter $P\_SH\_OUT_2$ and the output port P_out_lsb are connected to the Bus-1 While ports $P\_SH\_I_1$, $P\_ADD\_I_1$, P_out_msb, P_ADD_OUT, $P\_SH\_OUT_1$ and M_Reg are connected using Bus-2 The B_Reg is directly connected to the second input of the adder This is an instance of a point to point interconnection The main controller remains the same as in the previous example The local controller receives the control signals from the main controller and generates its own signals to carry out the necessary data transfers as shown in the Table 5 2a and 5 2b As can be seen from the tables, many data transfers are concurrent, thus increasing the speed of execution

Main con signals	Local con signals	Operation on	
		Bus 1	Bus 2
Start	Start	$A\_Port \longrightarrow A\_Reg$	$0 \longrightarrow M\_Reg$
Add	Add	–	$M\_Reg \longrightarrow P\_ADD\_I_1$
	Add2	–	$P\_ADD\_OUT \longrightarrow M\_Reg$
Shift	Shift	$A\_Reg \longrightarrow P\_SH\_I_2$	$M\_Reg \longrightarrow P\_SH\_I_1$
	Shift2	$P\_SH\_OUT_2 \longrightarrow A\_Reg$	$P\_SH\_OUT_1 \longrightarrow M\_Reg$
Done	Done	$A\_Reg \longrightarrow P\_out\_lsb$	$M\_Reg \longrightarrow P\_out\_msb$

Table 5 2a Data transfers in Multi-bus Shift Multiplier

Main controller signals	Local controller signals	Operation
		Non-bus
Start	Start	$B\_Port \longrightarrow B\_Reg$
Add	Add	$B\_Reg \longrightarrow P\_ADD\_I_2$
	Add2	–
Shift	Shift	–
	Shift2	–
Done	Done	–

Table 5 2b Data transfers in Multi-bus Shift Multiplier

Time Step	Operation	Dependencies
1	V_1	–
	V_2	–
	V_{10}	–
2	V_3	V_1, V_2
	V_6	V_2, V_3
	V_{11}	V_{10}
3	V_4	V_3
	V_7	V_4, V_6
	V_8	V_6, V_7
4	V_5	V_4, V_7
	V_9	V_8, V_{11}

Table 5 3 Operation dependencies in Differential Equation Integrator

5.3.3 Example 3 : Differential Equation Integrator

Consider the DFG scheduled in four time steps as shown in Figure 5 2 Let all the four datapath elements and the six registers used for storing three constants and three variables be connected to a single bus Further, assume that, all the eleven operations in the DFG take the same amount of time to execute In such a situation, execution of operations corresponding to any time step can be ordered to improve performance

Table 5 3 shows the data dependency of each operation on the other operations in the DFG These dependencies can be easily determined from the DFG Assume that a datapath element DPE_m has completed its current operation V_i Let the result of V_i be used to initiate another operation V_j Also assume that the next operation to be executed by DPE_m is V_k Then a temporal constraint needs to be imposed on the execution of V_k which can not begin until V_i has transferred its output to V_j

▮ *Ordering of Operations*

1 The various operations corresponding to a single time step are first ordered according to the dependencies that they have to follow Therefore, in time

step 2, V_3 should be executed before V_6 Sımılarly, ın tıme step 3, ordered sequence of executıons ıs V_4, V_7 and then V_8

2 Usıng the orderıng carrıed out ın step 1, the remaınıng operatıons can be ordered to ımprove performance

  (a) As V_4 ıs executed before V_8 ın tıme step 3, V_5 should be executed before V_9 ın tıme step 4

  (b) Executıon of V_7 depends on V_6 Therefore, V_6 ıs executed before V_{11} ın tıme step 2

  (c) Executıon of V_3 ın tıme step 2 depends on the executıon of V_1 and V_2 Therefore, ın tıme step 1, V_1 and V_2 are executed before V_{10} It can be noted that V_1 and V_2 can be executed ın any order

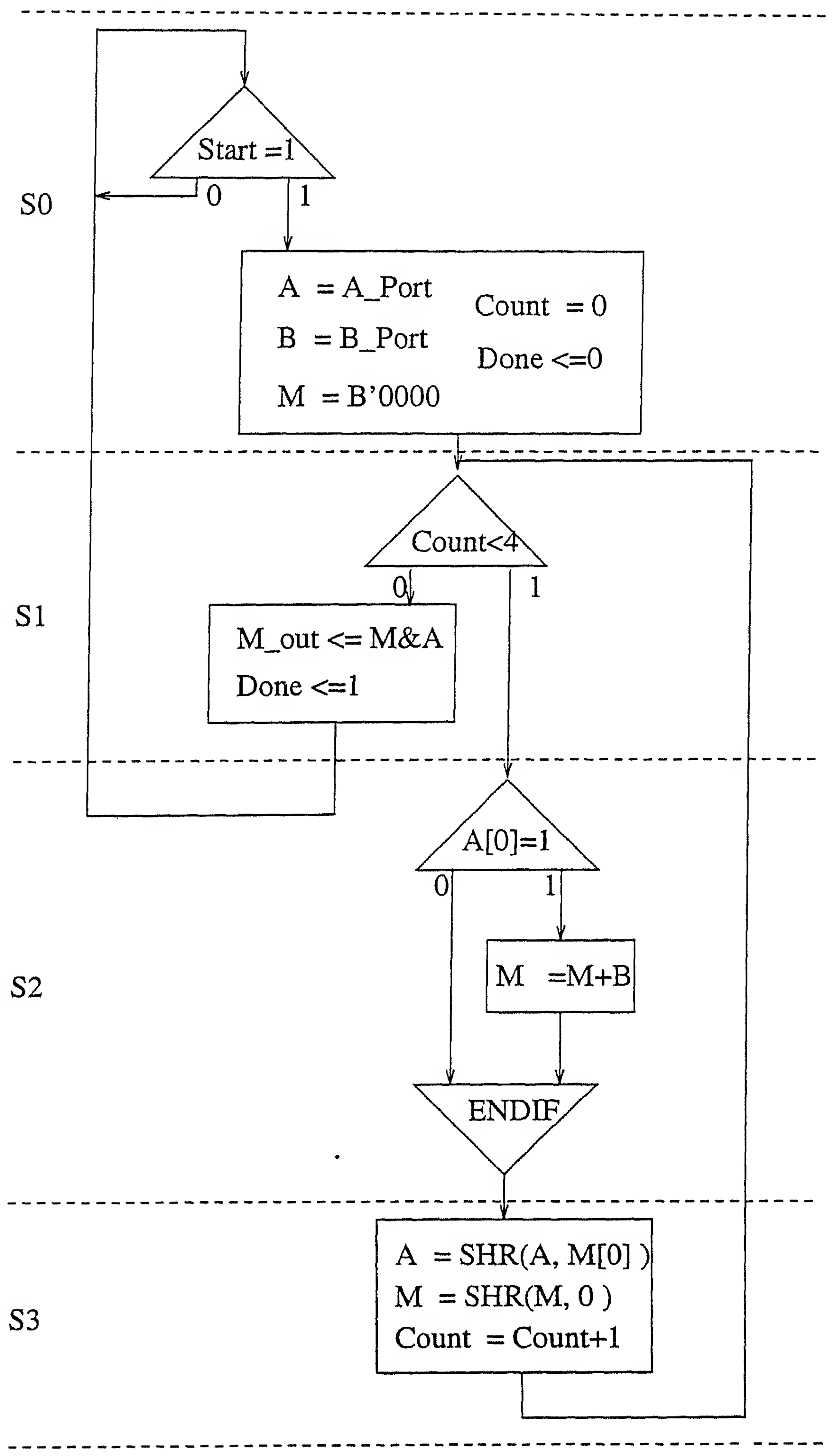


Figure 5 6 CDFG of the Shift Multiplier

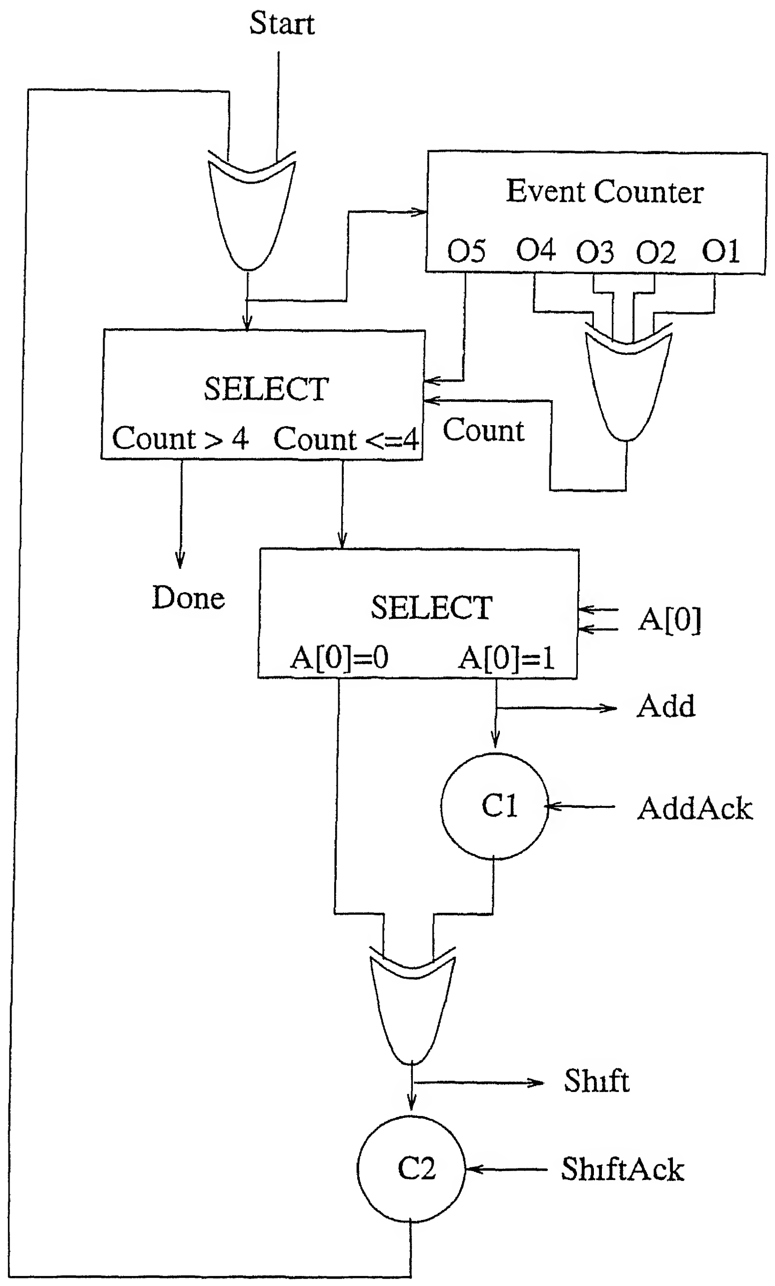


Figure 5 7a Shift Multiplier(Pt to Pt) Main Controller

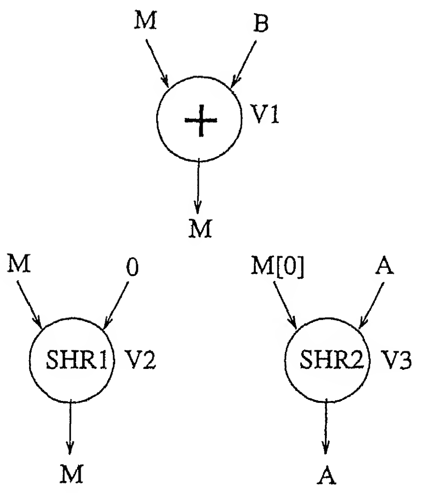


Figure 5 7b DFG of the Shift Multiplier

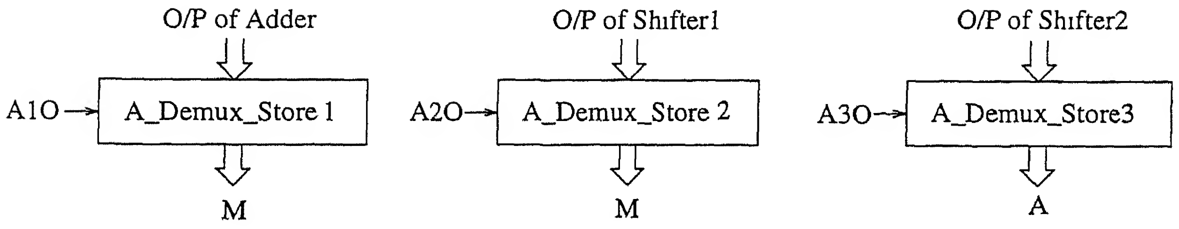


Figure 5 7c A_Demux_Store Elements

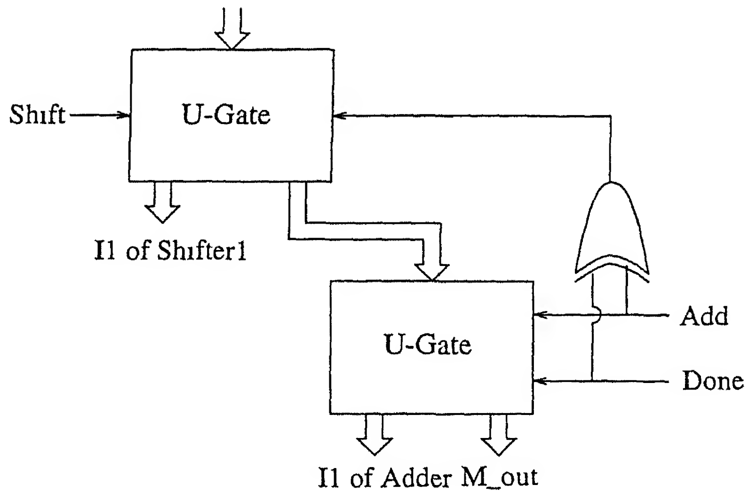


Figure 5 7d Iter_Var_Store element for M

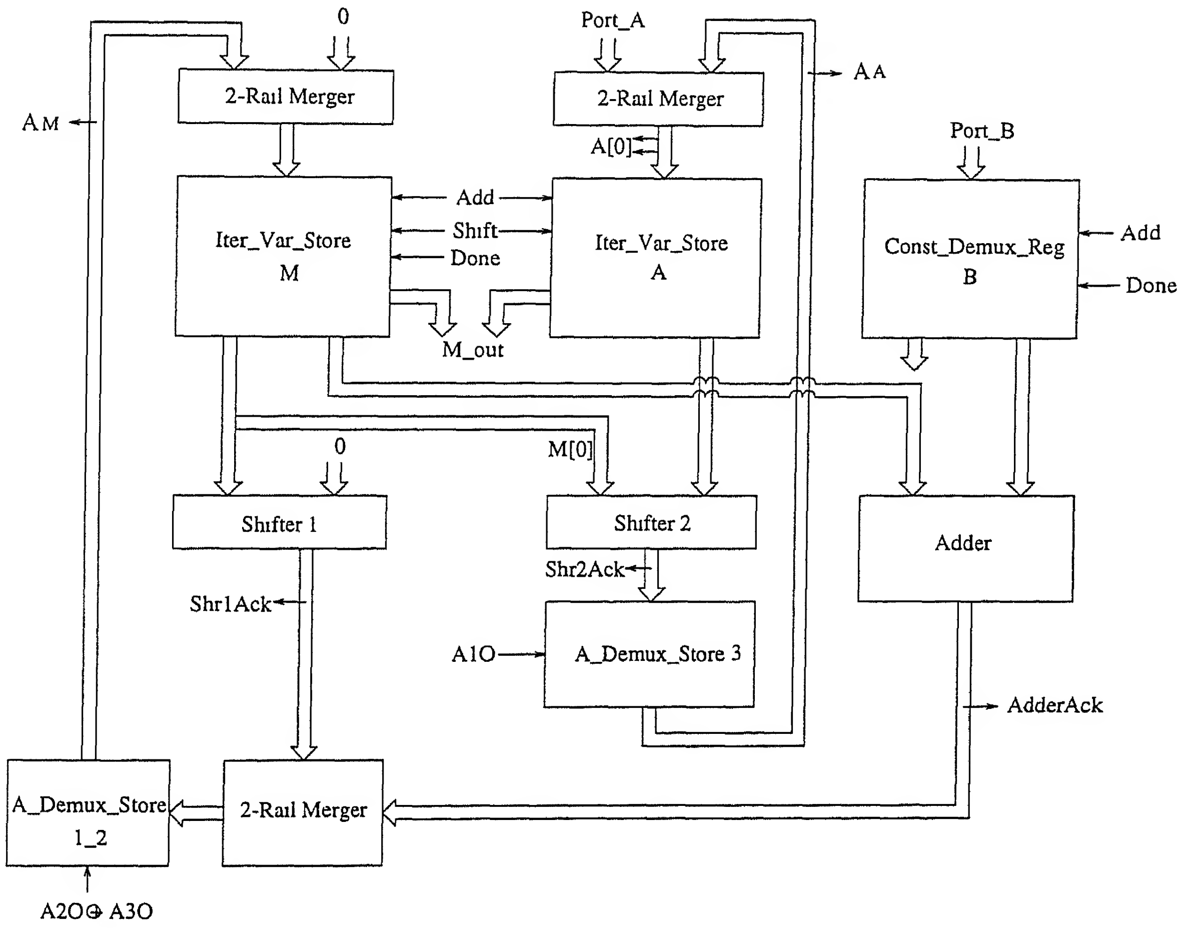


Figure 5 7e Shift Multiplier(Pt to Pt)   Datapath

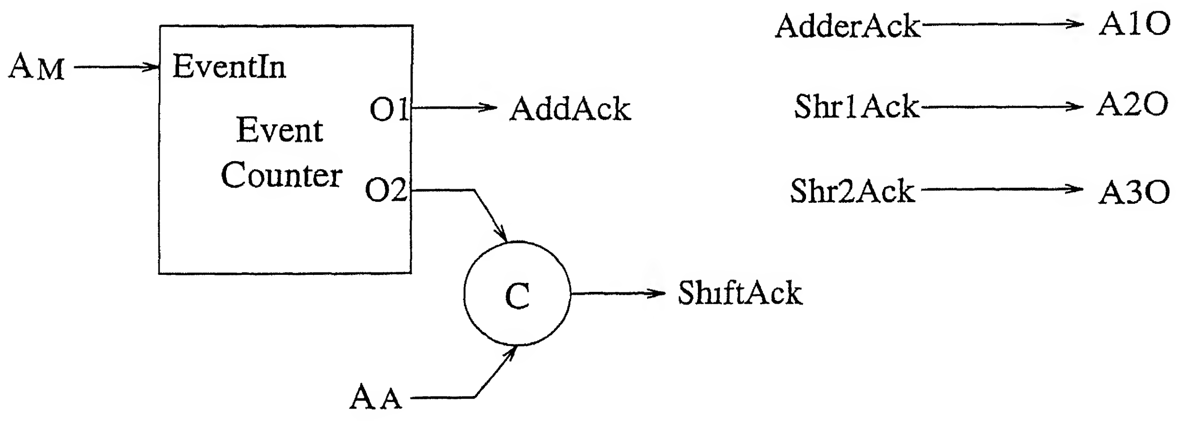


Figure 5 7f Shift Multiplier(Pt to Pt)   Local Controller

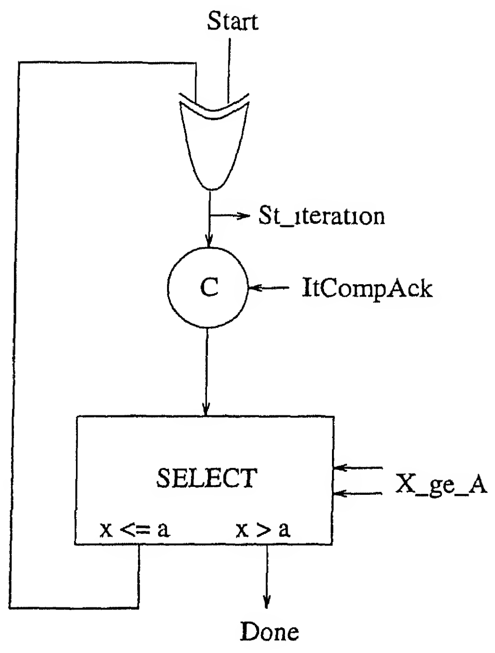


Figure 5 8a Differential Equation Integrator   Main Controller

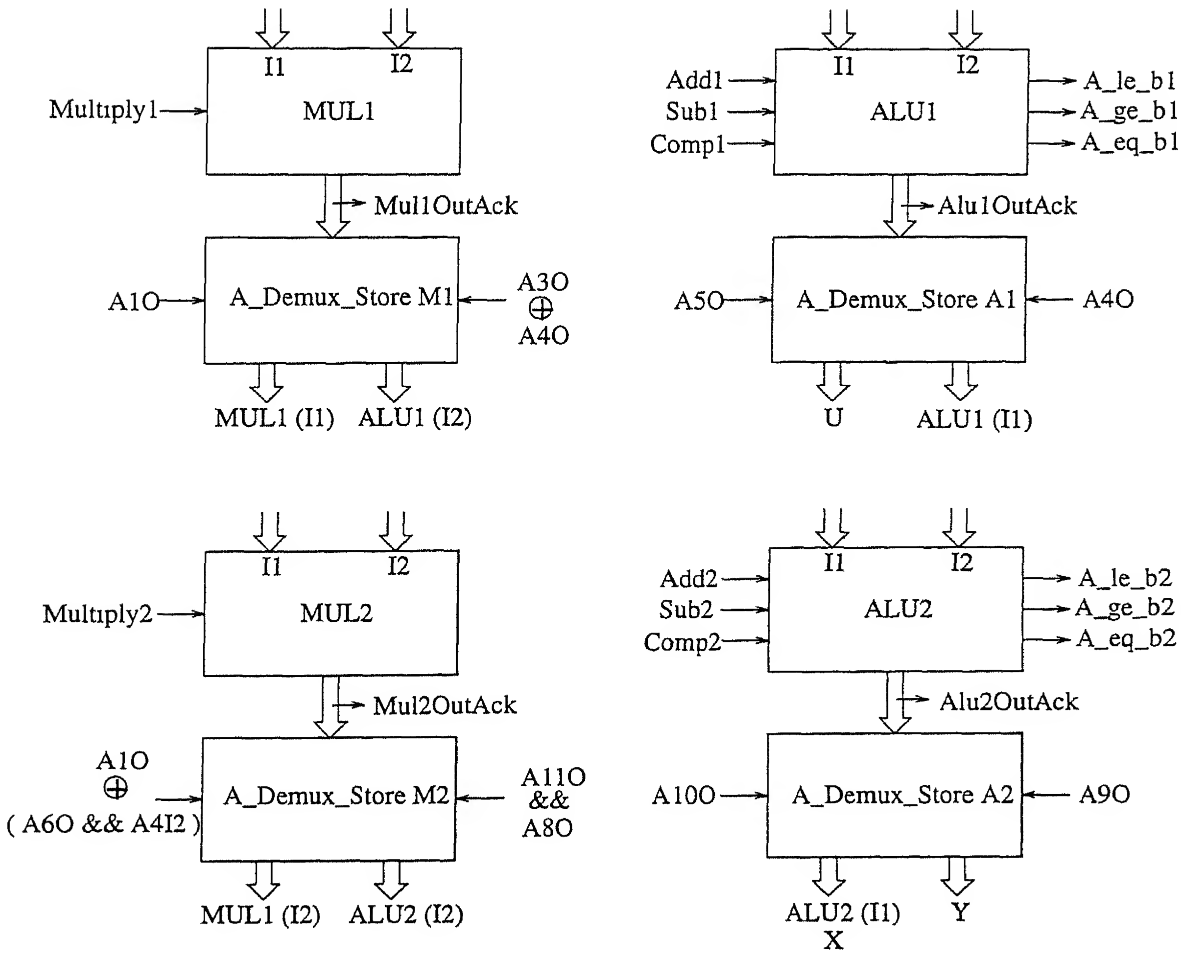


Figure 5 8b Differential Equation Integrator A_Demux_Store and Datapath Elements

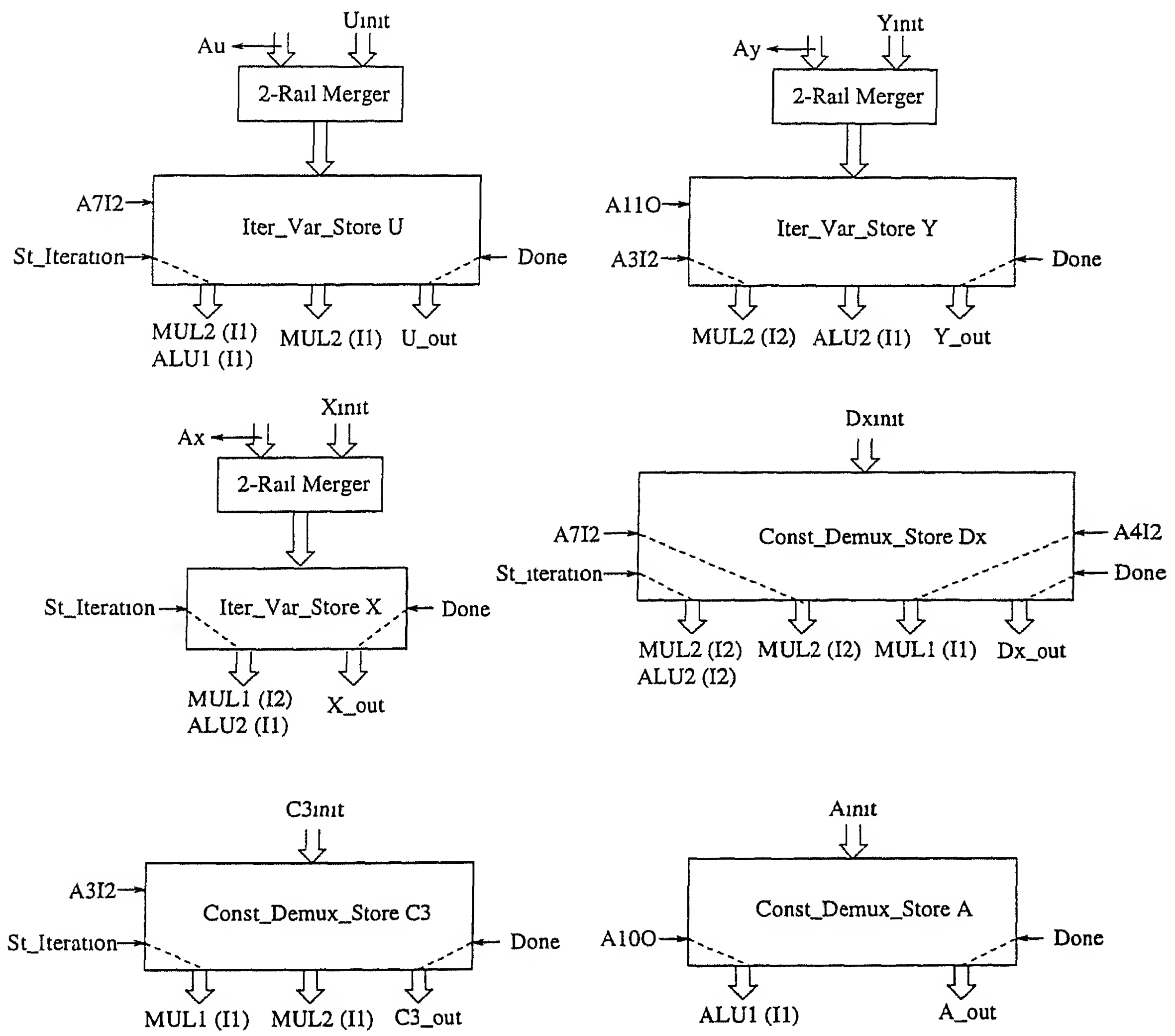


Figure 5 8c Differential Equation Integrator Registers

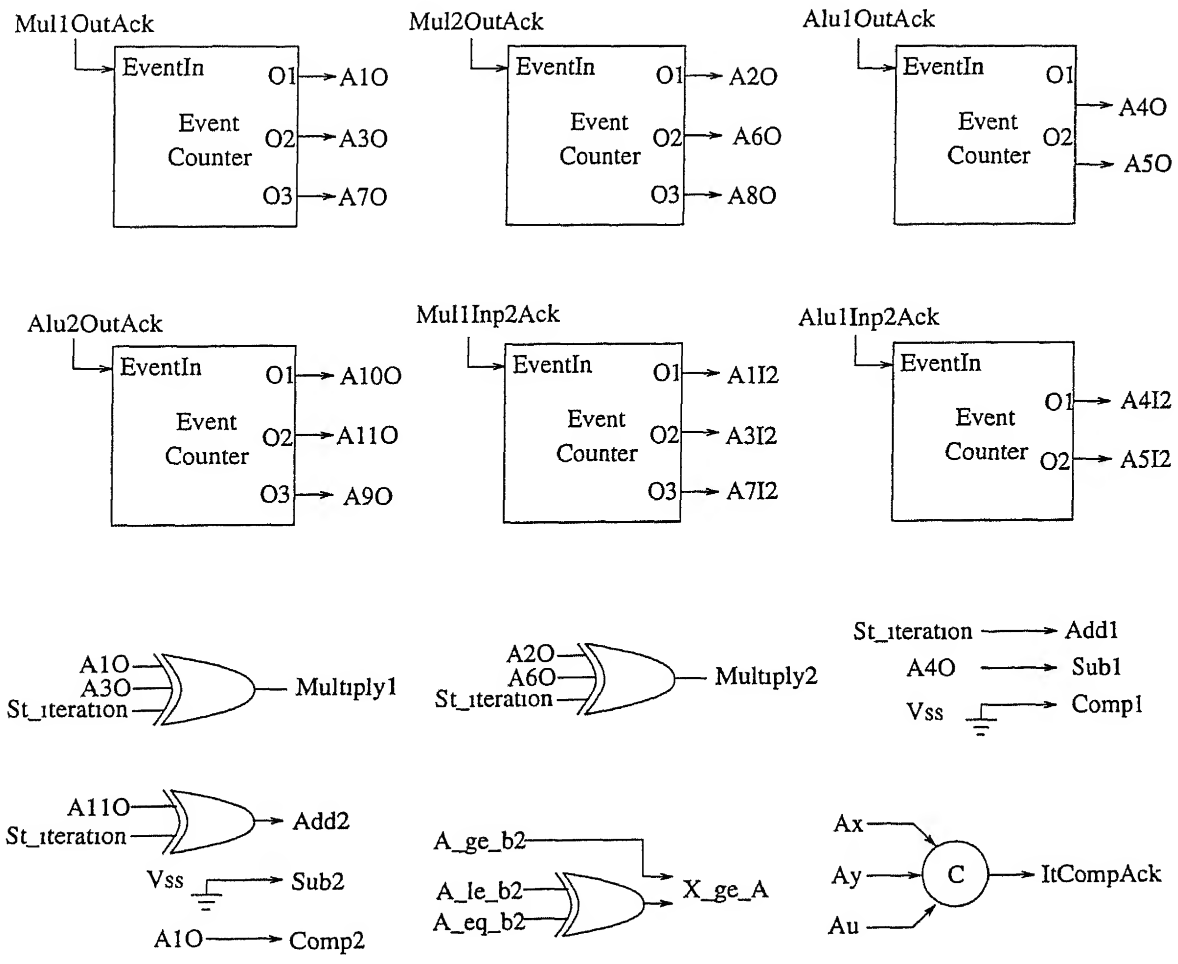


Figure 5 8d Differential Equation Integrator Local Controller

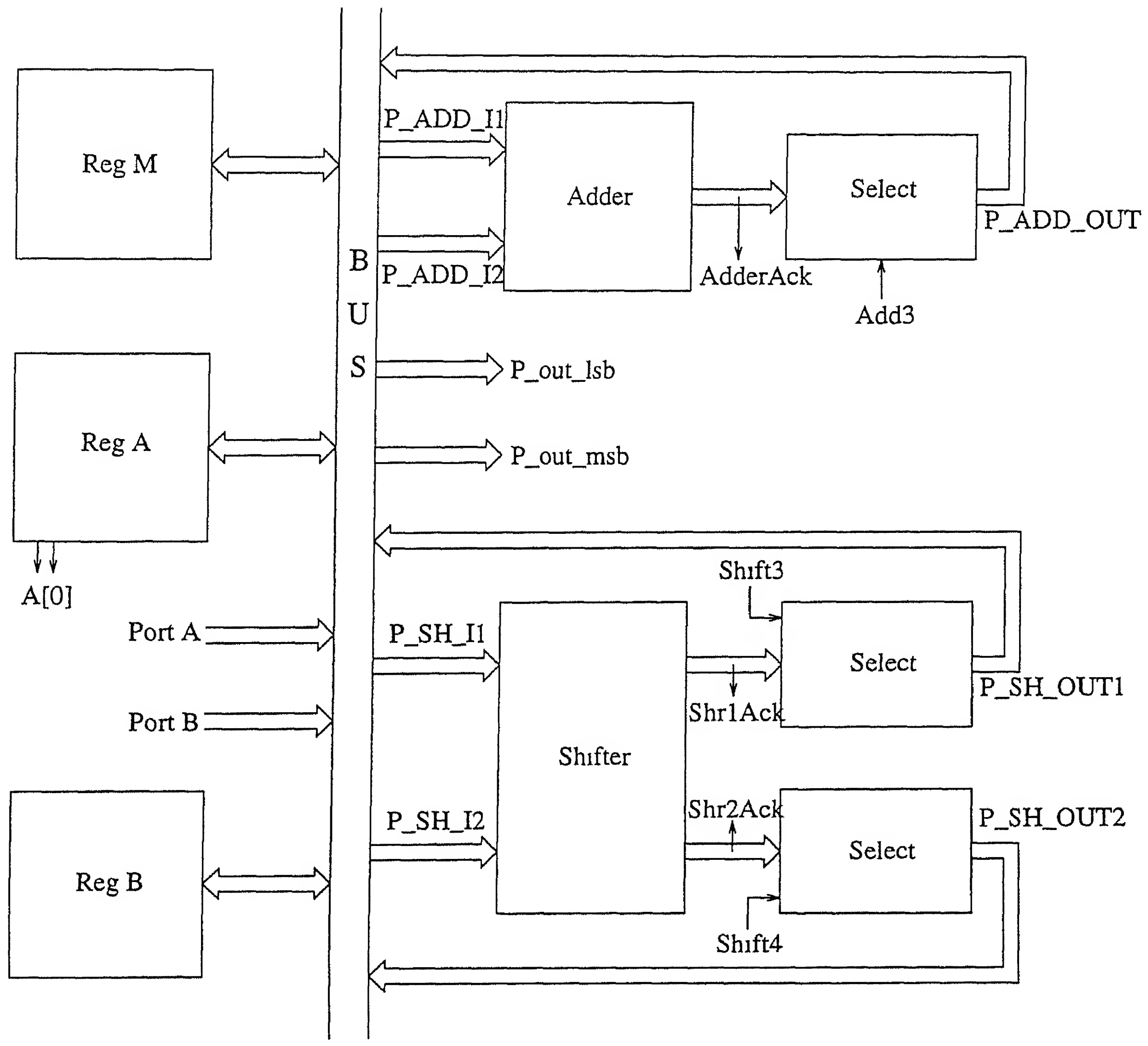


Figure 5 9a Shift Multiplier(Single-bus) Datapath

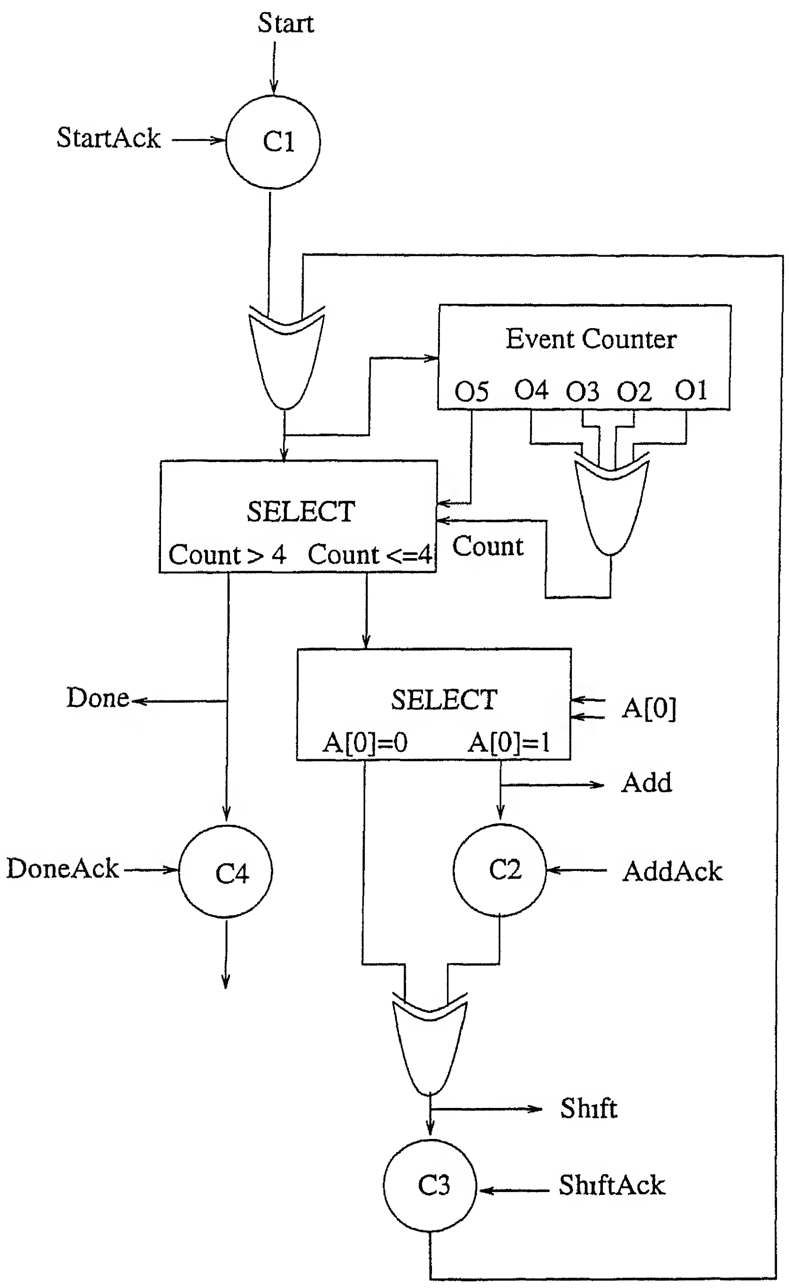


Figure 5 9b Shift Multiplier(Single-bus) Main Controller

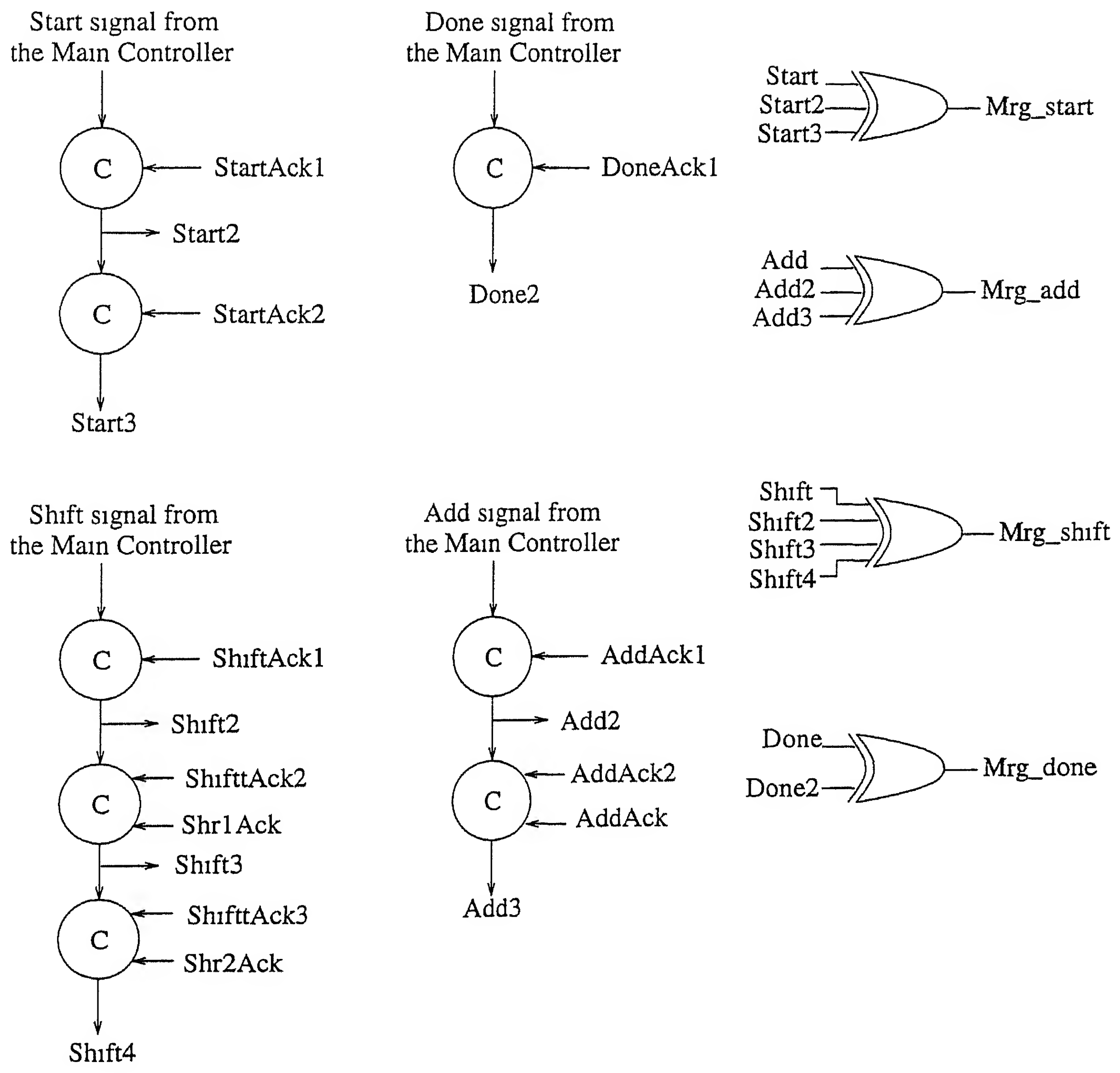


Figure 5 9c Shift Multiplier(Single-bus) Local Controller

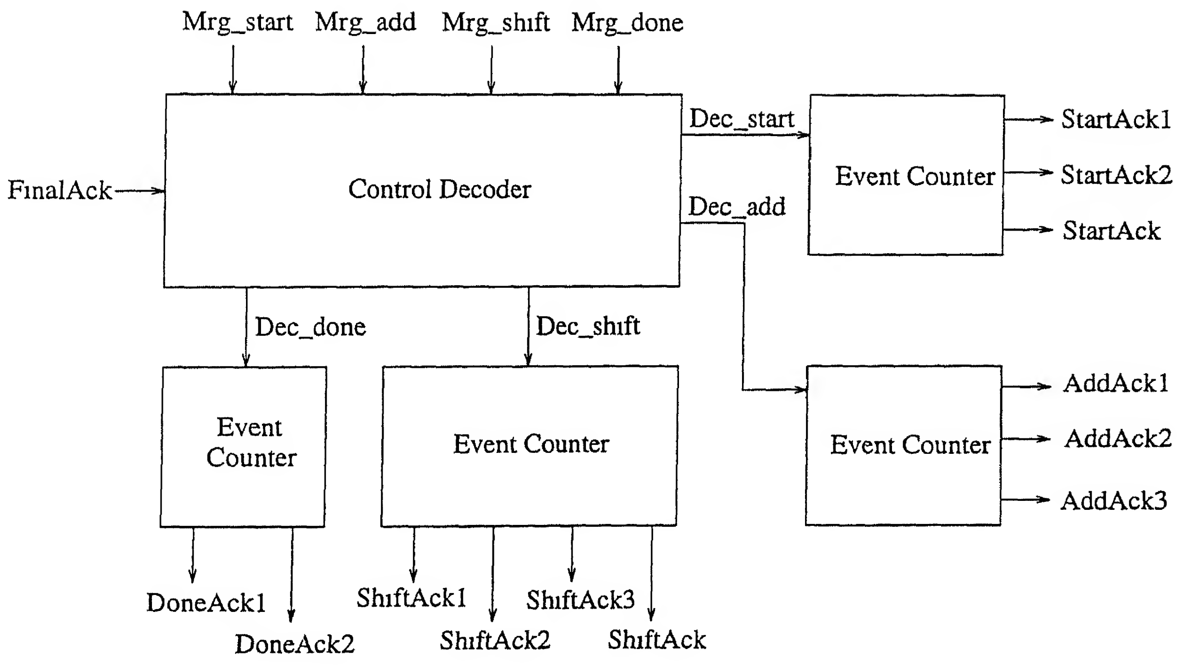


Fıgure 5 9d Shıft Multıplıer(Sıngle-bus) Local Controller

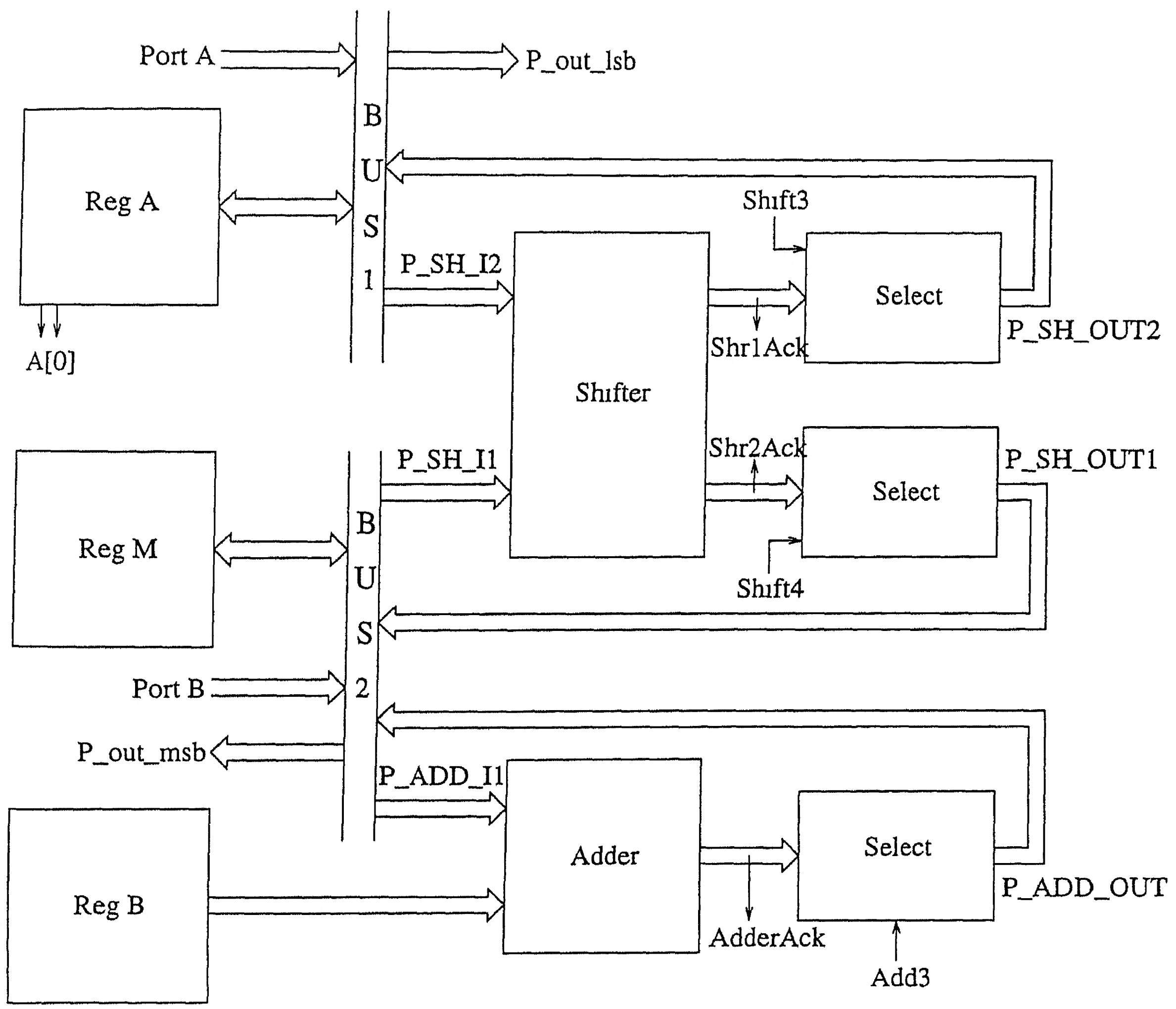


Fıgure 5 10a Shıft Multıplıer(Multı-bus) Datapath

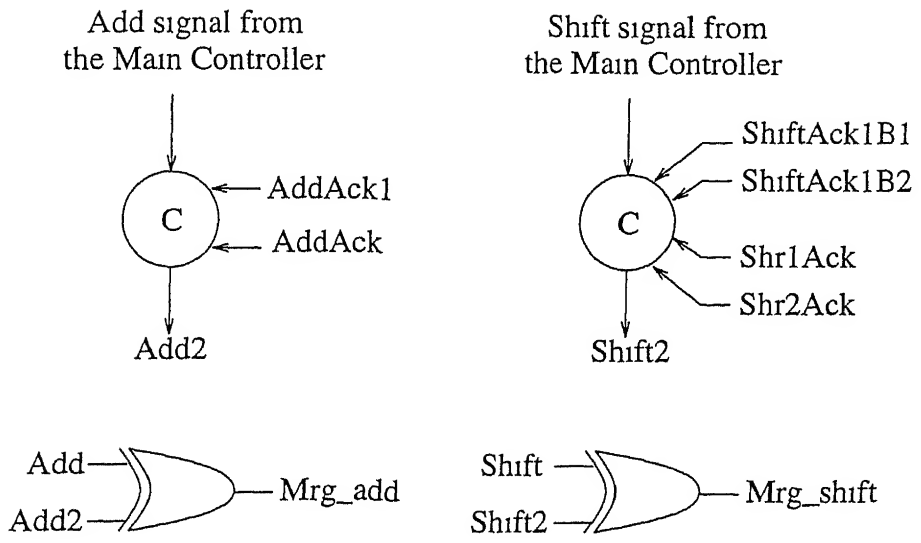


Figure 5 10b Shift Multiplier(Multi-bus) Local Controller

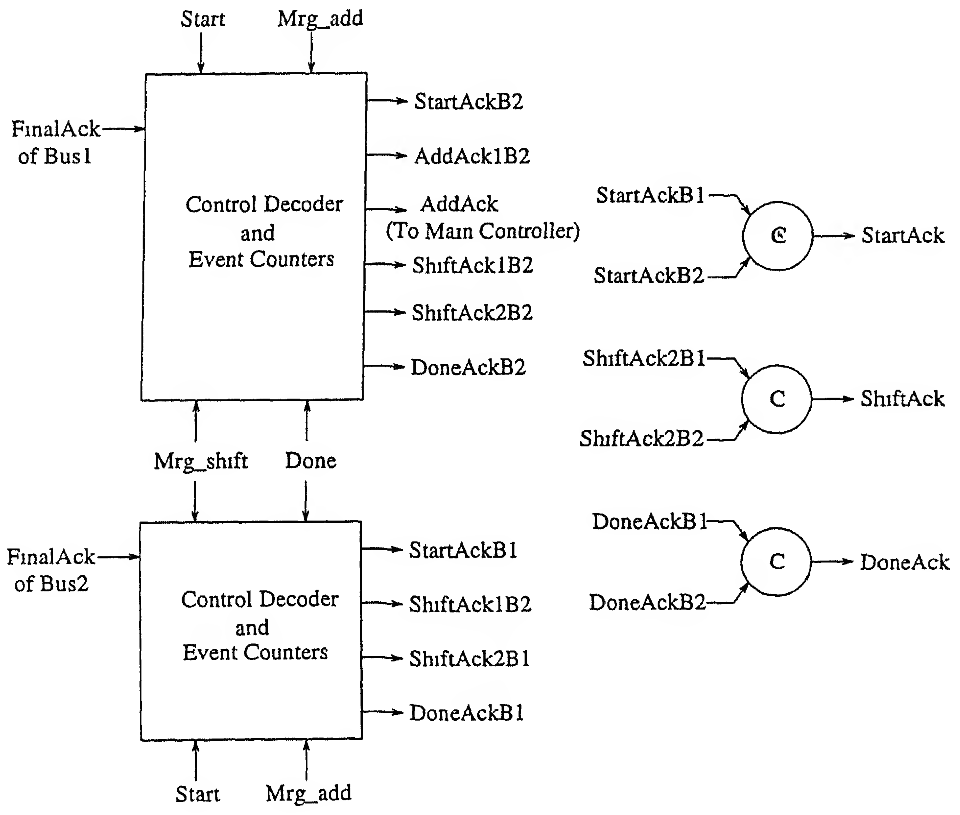


Figure 5 10c Shift Multiplier(Multi-bus) Local Controller

Chapter 6

Conclusion and Future Work

Conclusion :

Design of basic modules to systematically realise and synthesize implementations for asynchronous systems using the 2-phase NRZ transition signalling protocol has been given The use of these modules has been illustrated with three design examples, viz Shift Multiplier, Polynomial Serial Parallel Multiplier and Counter Interconnection schemes based on both the point to point topology and the bus structure have been given Three approaches to implement the bus structure have been described Finally, synthesis issues have been discussed The synthesis procedure outlined has been justified through two examples, viz Shift Multiplier and Differential Equation Integrator

Future Work :

In the synthesis approach, based on the bus topology, all data transfers needed to execute a set of operations, have to be implicitely sequential This can work well with the smaller designs However, for larger designs distributed over many chips, it can be a major restriction Synthesis for such designs should allow data transfer between different circuit blocks on different chips as and when they are ready for the data transfer to take place This necessitates the use of arbitration to resolve conflicts arising out of concurrent requests for any common resource, such as a bus To take care of this, we need to develop suitable arbitration logic based on our signalling protocol

Three approaches have been discussed in [9, 10] to realise Boolean expressions using U gates None of these can be shown to lead to optimal expressions A multilevel logic optimization method needs to be formulated to exploit the basic property of the U gate, in that it generates all the 2^n minterms with respect to n inputs

Appendix A

Simulations - SPICE and VERILOG

A detailed simulation using SPICE, at the transistor level, has been carried out for the C element and XOR gate Using typical process parameters for a 1 μm technology, from the SPICE simulations the average case delays with respect to $100 fF$ load capacitor (50 unit loads, where 1 $ul = 2\ fF$) for the C element and XOR gates implemented using minimum feature size transistors are found to be 1 8 nS and 1 1 nS respectively Similarly typical delay for a UReg based on SPICE simulations is 3 nS

The delays for the basic elements obtained through SPICE are used in the Verilog model for any design implementation Specifically, simulations in Verilog have been carried out using the above delays for the structural models of the implemented designs All the basic modules and the designs described in this thesis have been simulated in Verilog except for the designs described in the last section of chapter 5 i e , design examples based on the bus interconnection topology

Some of the simulation results are listed below

1 For the 4-bit Shift Multiplier implemented using the point to point interconnection topology, the worst case delay (multiplying decimal 15 by 15) is found to be approximately 300 nS While the best case delay (multiplying decimal 15 by 0) is found to be approximately 136 nS The worst case and the best

case delays are not in relation to the parametric delays but with respect to data dependent delays The corresponding synchronous implementation will take 14 clock cycles to execute the multiplication for all the possible cases

2 The Differential Equation Integrator has been simulated using the behavioral modules of the ALU and the 4-bit multiplier The delay of 25 nS was assigned to the ALU and that of 200 nS to the multiplier The delay in executing a single iteration is found to be 685 nS

3 The Polynomial Serial Parallel Multiplier has a delay of approximately 50 nS for each serial output bit produced i e , the throughput for PSPM is about 50 nS

The behavioral and the structural descriptions in Verilog have been created for all the modules and can be invoked as basic library elements for creating new design implementations and simulating them

While simulating every design implementation, a test for the delay-insensitivity was conducted by assigning arbitrarily large and random delays to different modules and interconnections In every case, the corresponding implementation worked correctly thus proving that this methodology is truly delay-insensitive

Bibliography

[1] Scautt Hauck, "Asynchronous desıgn methodologıes An overvıew", Tech Rep 93-05-07, Department of Computer Scıence and Engg , Unıversıty of Wahıngton, Seattle, 1993

[2] S H Unger, *Asynchronous Sequentıal Swıtchıng Cırcuıts*, Wıley-Interscıence, New York, 1969

[3] S H Unger, "Hazards, crıtıcal races and metastabılıty", *IEEE Transactıons on Computers*, vol 44, no 6, pp 754–768, June 1995

[4] A J Martın, "The lımıtatıons to delay ınsensıtıvıty ın asynchronous cırcuıts", ın *6th MIT Conf on Advanced Research ın VLSI*, Cambrıdge, MA M I T Press, 1990, pp 263–278

[5] J C Ebergen, "A formal approach to desıgnıng delay-ınsensıtıve cırcuıts", *Dıstrıbuted Computıng*, vol 5, no 3, pp 107–119, July 1991

[6] J A Brzozowskı and J C Ebergen, "On the delay-sensıtıvıty of gate networks", *IEEE Transactıons on Computers*, vol 41, no 11, pp 1349–1360, November 1992

[7] S C Leung and H F Lı, "On the realızabılıty and synthesıs of delay-ınsensıtıve behavıors", *IEEE Transactıons on Computer Aıded Desıgn of Integrated Cırcuıts and Systems*, vol 14, no 7, pp 833–848, July 1995

[8] F U Rosenberger, C E Molnar, T J Chaney, and T P Fang, "Q modules Internally clocked delay-ınsensıtıve modules", *IEEE Transactıons on Computers*, vol 37, no 9, pp 1005–1018, September 1988

[9] K Nanda, *Desıgn of Asynchronous Dıgıtal Cırcuıts*, B Tech Report, Department of Electrıcal Engg , Indıan Instıtute of Technology, Kanpur, Aprıl 1996

[10] K Nanda, S K Desaı, and S K Roy, "A new methodology for the desıgn of asynchronous dıgıtal cırcuıts", ın *10th Internatıonal Conference on VLSI Desıgn*, Hyderabad, Indıa, 1997, pp 342–347

[11] T Nanya, Y Ueno, H Kagotanı, M Kuwako, and A Takamuro, "Tıtac Desıgn of a quası-delay-ınsensıtıve mıcroprocessor", *IEEE Desıgn and Test of Computers*, pp 50–63, Summer 1994

[12] J E Sutherland, "Mıcropıpelınes", *Communıcatıons of the ACM*, vol 32, pp 720–738, June 1989

[13] A J McAuley, "Four state asynchronous archıtectures", *IEEE Transactıons on Computers*, vol 41, no 2, pp 129–142, February 1992

[14] D Gajskı, N Dutt, A Wu, and S Lın, *Hıgh Level Synthesıs Introductıon to Chıp and System Desıgn*, Kluwer Academıc Publıshers, 1992

[15] Gıovannı De Mıchelı, *Synthesıs and Optımızatıon of Dıgıtal Cırcuıts*, Mc-Graw Hıll, New York, 1994

[16] K Maheshwaran and Venkatesh Akella, *Hazard-Free Implementatıon of the Self-Tımed Set for the Xılınx 4000 Serıes FPGA*, Department of Electrıcal Engg , Unıversıty of Calıfornıa, Davıs, CA 95616, Prıvate Communıcatıons.

[17] C L Seıtz, "System tımıng", ın *Introductıon to VLSI Systems*, C Mead and L Conway, Eds Readıng, MA Addıson-Wesley, 1980, pp 218–262

EE-1997-M-DES-DES